RL-TR-97-95
Final Technical Report
September 1997

THE THETA (TRUSTED HETEROGENEOUS ARCHITECTURE SYSTEM (VERSION 2.2)

Odyssey Research Associates, Inc.

Bret Hartman, Cheryl Barbasch, Matthew Stillerman,
Rita Pascale, Joseph McEnerney, Michael Seager,
David Guaspari, David Balenson and Maureen Stillman

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-95 has been reviewed and is approved for publication.

APPROVED: Emilie J Siarkiewicz

EMILIE J. SIARKIEWICZ
Project Engineer

FOR THE DIRECTOR: John A Graniero

JOHN A. GRANIERO, Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3AB, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Sep 97	Final Jul 92 - Dec 95

4. TITLE AND SUBTITLE

THE THETA (TRUSTED HETEROGENEOUS ARCHITECTURE (VERSION 2.2)

5. FUNDING NUMBERS

C - F30602-92-C-0145
PE - 33140F/33401G
PR - 7820
TA - 04
WU -10

6. AUTHOR(S)

Bret Hartman, Cheryl Barbasch, Matthew Stillerman, Rita Pascale, Joseph McEnerney, Michael Seager, David Seager, David Guaspari, David Balenson and Maureen Stillman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

PRIME
Odyssey Research Associates, Inc.
301 Dates Dr.
Ithaca, NY 14850

SUB
Trusted Information Systems, Inc.
3060 Washington Rd (Rt 97)
Glenwood, MD 21738

8. PERFORMING ORGANIZATION REPORT NUMBER

ORA TM-95-0023

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3AB
525 Brooks Rd.
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-95

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Emilie J. Siarkiewicz, C3AB, 315-330-2135

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report is intended for people who wish to acquire a general conceptual overview of the THETA system and the technical advancements that resulted from the research. The report defines the terminology and the basics of secure, distributed, object-oriented environments. Programming for the THETA system, administration, usage, and other related information is provided in other contract deliverables listed in this report. Chapter 1 contains an introduction to THETA, explaining its security philosophy, architecture, and implementation. Chapter 2 provides a comparison of THETA and CORBA, and discusses the potential for a CORBA-compliant version of THETA. Chapter 3 compares THETA and DCE, and explores approaches to integration. Chapter 4 presents a list of the accomplishments of the THETA project. Lessons learned from this effort are presented in Chapter 5. Possibilities for future THETA development are outlined in Chapter 6.

14. SUBJECT TERMS

Multilevel Security, Object-oriented, Distributed Processing

15. NUMBER OF PAGES

126

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NOTICES

The following products are trademarks or registered trademarks of their respective companies or organizations:

HP-UX and HP-UX BLS 8.09+ by Hewlett-Packard Corporation
SunOS and SunOS CMW by Sun Microsystems, Inc.
UNIX by AT&T, Inc.
Cronus by BBN Systems and Technologies

Table of Contents

List of Figures

Preface

This document includes an introduction to THETA, compares THETA to other distributed computing approaches, and describes some lessons learned from the current THETA development. The document is intended to be a high-level overview. Where more detail is necessary, pointers to the appropriate support documentation are provided.

The *Final Report for THETA* is intended for people who wish to acquire a general conceptual overview of the THETA system and the technical advancements that resulted from the research. The report defines the terminology and the basics of secure, distributed, object-oriented environments. Programming for the THETA system, administration, usage, and other related information is provided in the supporting documentation that is listed below.

Organization of This Document

Chapter 1 contains an introduction to THETA, explaining its security philosophy, architecture, and implementation. Chapter 2 provides a comparison of THETA and CORBA, and discusses the potential for a CORBA-compliant version of THETA. Chapter 3 compares THETA and DCE, and explores approaches to integration. Chapter 4 presents a list of the accomplishments of the THETA project. Lessons learned from this effort are presented in Chapter 5. Possibilities for future THETA development are outlined in Chapter 6.

THETA Documentation Set

THETA is a large system with many facets. The documentation set is split into logical parcels as follows:

- *Introduction to THETA* - This document describes the THETA system, its architecture and capabilities. It is intended to be a high-level overview.
- *Other Documents* - The documents listed below contain further details on specific topics.

- *Manager Developer's Tutorial* - This two volume document leads the programmer through the steps involved in developing, testing, debugging, and maintaining a manager. Volume II details the manager generation process.
- *Software User's Manual* (SUM) - This document describes how to interact with THETA in a consistent and coherent manner. "Tropic" is the main application discussed in this document.
- *Computer System Operator's Manual* (CSOM) - This two part document defines the role of the THETA operator and describes administrative duties such as starting, maintaining, and stopping the THETA system; the second part is the installation guide for THETA, which describes the general concepts of setting up THETA on a host and then describes the different steps needed on each supported platform.
- *Software Programmer's Manual* (SPM) - This three volume document is a reference for THETA programmers.
- *System Segment Specification* (SSS) - This document discusses the various components of the THETA system and their respective functions.
- *System Segment Design Document* (SSDD) - This document lays out the design and proposed implementation of the components described in the *System Segment Specification.*
- *Software Requirements Specification* (SRS) - This three volume document states the software engineering requirements of the various THETA components.
- *Interface Requirements Specification* (IRS) - This document outlines the various requirements of the interfaces to the trusted computing base of the THETA system.
- *Software Design Documents* (SDD) - This collection of documents provides details on every software component of the THETA system down to the level of pseudo code.
- *Interface Design Document* (IDD) - This document describes the details of the various THETA kernel interfaces.
- *Software Development Plan* (SDP) - This document describes the software development cycle of THETA and its various components; it also details the software engineering techniques used to ensure code correctness and maintainability.
- *Version Description Document* (VDD) - This document specifies the version number of THETA in terms of its functionality, platform availability, known problems, and future work planned.
- *Formal Security Policy Model* (FSPM) - This document contains the THETA security policy and a formal model of that policy. It also states how the formal model clearly maps to the actual THETA implementation.
- *Philosophy of Protection Report* (POP) - This document contains the assurance arguments used to support THETA's claim of B3 compliance.

- *Descriptive Top Level Specification* (DTLS) - This document describes the various THETA components at a high level.
- *Covert Channel Analysis Report* (CCA) - This document contains the results of covert channel analysis of the THETA system and its individual components.
- *Trusted Computing Base Configuration Management Plan* (TCBCMP) - This document describes the configuration management plan for the trusted computing base code. This configuration management plan is an important software engineering step that facilitates version identification.
- *Trusted Computing Base Verification Report* (TCBVR) - This document describes the formal methods used in designing the THETA kernel and the mechanisms used to implement the design.
- *Security Test Plan* (STP) - This document outlines the various tests that have been run on the THETA system to validate that the mandatory and discretionary access control policies are being enforced.

1 THETA Introduction

1.1 Executive Summary

This report provides an introduction to the Trusted Heterogeneous Architecture, which is commonly known as THETA. This section gives a condensed description of THETA and its key features. The remainder of the chapter explains the THETA philosophy, architecture, and implementation in much greater detail.

1.1.1 System Overview

THETA (Trusted HETerogeneous Architecture) is a distributed, heterogeneous, secure operating system. It qualifies as an operating system because it controls access to the basic resources of a computer system. THETA is a *distributed* system since it allows access to resources that are spread across a network of systems. THETA handles the necessary translations between disparate operating systems and hardware architectures to permit seamless interoperability to the user and developer; thus, THETA is *heterogeneous*. THETA also enforces flexible, global access control policies over the network in accordance with *Trusted Computer System Evaluation Criteria* (TCSEC [4]) class B3; thus, THETA is *secure*.[1]

All of these features are easily provided because the system is object oriented. The complexities of security and heterogeneity are abstracted so that users can access the distributed resources in a coherent, uniform manner. A disparate set of hardware machines can be networked together, each contributing their own unique qualities and capabilities to the overall resource pool, and THETA provides the mechanism to access these various systems in a consistent fashion.

THETA is under development at Odyssey Research Associates, Inc. (ORA). Work has been sponsored by the Air Force's Rome Laboratory since 1985. The design of THETA is based heavily on the design of Cronus, which is a distributed, heterogeneous operating system developed at BBN.

[1] Note that THETA has not undergone the official security evaluation process; however, the system has been designed to meet the TCSEC B3 criteria.

1.1.2 Features

Features of THETA are highlighted below.

1.1.2.1 Coherent and Uniform

The THETA system provides a coherent method of accessing and manipulating distributed processing resources. System services are available to the user through a uniform set of abstractions. Objects such as files, directories, processes, services, and I/O devices are referenced through a global naming facility and a common set of communication primitives.

1.1.2.2 Heterogeneous

Many distributed systems have evolved through the interconnection of existing stand-alone machines of possibly different hardware and software architectures. These machines may be connected by a local-area network (LAN) at a specific location or by a wide-area network connecting LANs at different locations. THETA facilitates interconnection among machines of differing architectures, which promotes the sharing of information and computing resources between organizations and increases reliability and availability of services.

1.1.2.3 Evolvable

THETA allows an organization to keep up with new technology without rendering the old systems obsolete. New hardware can be added into the THETA network as it becomes available with minimal effort. However, hardware that predates 32 bit architecture and dated operating systems are problematic. Sixteen bit architecture and a lack of support for modern OS services are high risk and costly to port. New operating systems can plug into the THETA network, but this requires a porting effort, mainly of the Theta kernel, of about six months.

1.1.2.4 Efficient

Distributed resources are more effectively used because more resources are made available to a larger group of users. Also, THETA services are synchronous so that time-intensive operations do not block requests and replies from other clients.

1.1.2.5 Available

Since THETA resources and services are distributed across several machines, probabilities of service availability are higher than on a single host. In cases when a few machines are disabled, the majority of the THETA system is still assessable; thus the outage may be totally invisible to the users.

1.1.2.6 Reliable

The THETA system maintains data integrity in spite of system failures. The THETA system allows database replication for essential data to provide fault-tolerance. Crucial information can be duplicated across several hosts in the network so in the case of system failures and network partitions, there is a greater chance that this crucial data is still accessible.

1.1.2.7 Scalable

The THETA system may be configured with different processing elements to accommodate a range of users and applications. THETA can incrementally expand, migrate, and mutate to meet the current demands for services and resources.

1.1.2.8 Extensible

No computer software system is ever complete; therefore, it is important to provide the means to extend THETA in a secure way. THETA provides tools to generate clients and servers using general purpose templates. These templates can be copied and tailored to serve new purposes. This sort of rapid prototyping is a core feature of THETA.

1.1.2.9 Secure

The THETA system has been designed to meet the TCSEC B3 functionality and assurance requirements.[2] THETA enforces a consistent mandatory security policy and discretionary access control policy; provides reliable identification and authentication of users and their processes; audits user and system activity; and provides other distributed security services. A high-level discussion of THETA security features is in Chapter 1.4; however, for more extensive detail, see documents [30], [31], [33], and [37] that are listed in the Section "References" on page 91.

1.1.2.10 Confidential Communication

As a distributed system, THETA must send messages and data over a network. To ensure security, these messages over the network are protected via encryption. This network security is provided by TNET. For details on the implementation and the extent of the protection provided, reference [39], [40], and [41].

---

[2] Although THETA has not been subjected to an official security evaluation, it has been designed to meet TCSEC B3 criteria. Discussion of assurance arguments to accompany the claim of B3 compliance appear in Section 1.5.5.

1.1.3 Available Platforms

THETA has been ported to an assortment of secure operating systems as well as some untrusted platforms. Currently supported THETA platforms are at TCSEC B1 level. Obviously, THETA does not meet the TCSEC B3 criteria when running on platforms built at levels lower than TCSEC B3. Porting THETA to a high assurance platform is a future goal of our effort. THETA is available on the following operating systems:

- HP-UX BLS 8.09+ - This system is designed to meet the TCSEC level B1. THETA 1.5 (excluding TNET) and 1.7b (excluding TNET) run on this system.
- HP-UX 8.09 - This system is an untrusted operating system and does not meet standards specified in TCSEC. THETA 1.5 (excluding TNET) and 1.7b (excluding TNET) run on this system.
- Sun CMW 1.0 - The Sun Compartmented Mode Workstation is designed to meet the Compartmented Mode Workstation requirements, which are similar to requirements for TCSEC B1 systems; however, Compartmented Mode Workstations have additional criteria concerning windowing environments. See [43] in "References" on page 91. THETA 1.5 (including TNET) and 1.7b (excluding TNET) run on this system.
- SunOS 4.1.X - This system is an untrusted operating system and does not meet standards specified in TCSEC. THETA 1.5 (including TNET), 1.7b (excluding TNET), and 2.3 (excluding TNET) run on this system..
- AT&T System V MLS - This system is designed to meet TCSEC B1 criteria. THETA 1.3a (excluding TNET) runs on this system.
- Trusted Solaris 1.2 - This system is designed to meet TCSEC B1 criteria. THETA 2.3 (excluding TNET) runs on this system.

1.2 Distributed Computing

1.2.1 Background

Computing technology has migrated from large, expensive, stand-alone machines to highly specialized networks of machines composed of workstations and personal computers that far surpass the processing power of their monolithic predecessors. The variety of hardware available today is vast and disparate. Machines differ in cost, speed, and functionality. Organizations must assess their needs to select the best mix of computing resources.

To maximize resource usage, machines are often networked together. Interconnections allow data sharing across hosts, which saves space and allows remote machines access to processing resources that would not be available in a stand-alone environment. These benefits do not come for free; while gaining data-sharing capabilities, networked systems lose ground in security. Sending data over a network creates more risks to data confidentiality, integrity, and availability. For more information on the dangers of networking and computing in general, see [28] in "References" on page 91.

Despite the risks of open-computing, the benefits of reduced operational costs and increased efficiency have caused the computing industry to press on into the world of distributed computing. Now, not only data is shared across platforms, but processing power is too. Today's computing environment consists of a network of cooperative data resources and processes.

Machines are becoming specialized to better perform their particular tasks; for example, there are super processing machines like Cray, graphics machines like Silicon Graphics, database machines like Teradata, and artificial intelligence machines like Symbolics. Many of these hardware architectures are targeted to perform a special function; however, some applications need a sampling of each machine's capabilities. Simply networking the machines together may not solve the processing requirements of some applications. One solution to using distributed resources is to create a single distributed application that performs a portion of its processing on each specialized machine.

1.2.1.1 The Need for Distributed Systems

The following example illustrates the need for distributed systems. This scenario requires several specialized hardware architectures to complete a single task. Scientists are trying to predict natural disasters in order to facilitate evacuations of the area. Weather monitoring instruments on satellites beam down massive amounts of data to collection centers where the raw data is processed by a supercomputer. The processed data is sent to specialized graphics stations as it becomes available and is also stored in a database machine for future reference. The scientists constantly monitor the data at the graphics stations. From there, the scientists analyze the information and send out hypothetical queries to an artificial intelligence machine. The artificial intelligence machine makes predictions of potential disasters and makes recom-

mendations on efficient evacuation procedures, which are relayed back to the scientists at the graphics station.

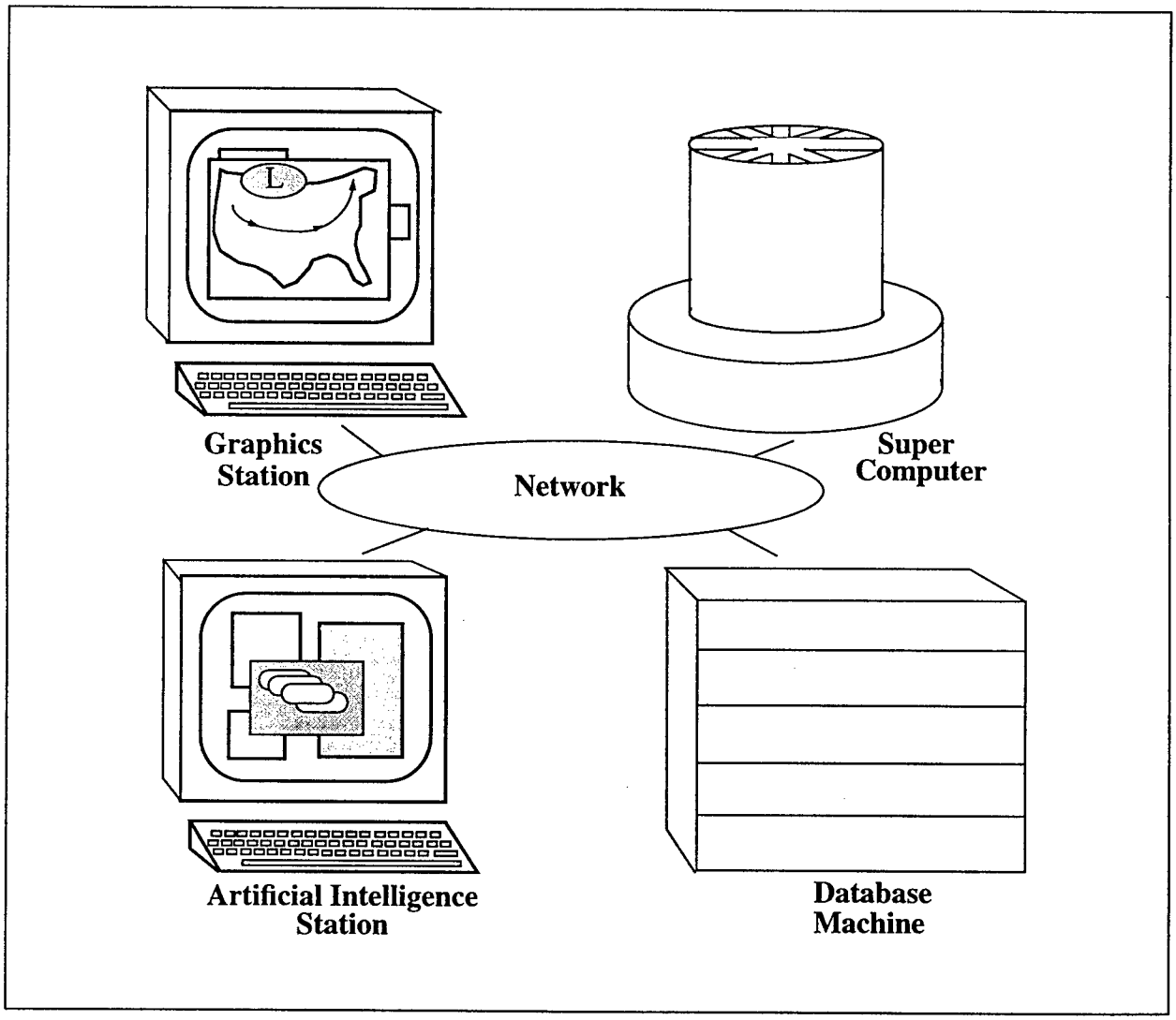

Figure 1-1: Sample Network for the Distributed Application Example

1.2.1.2 The Need for Secure Distributed Systems

Modifying the above example slightly, security becomes a serious requirement. Instead of weather satellites, data is pouring in from classified intelligence satellites. The scientists are replaced by intelligence analysts who are tracking international troop movements. The artificial intelligence system is used to predict potential military conflicts and provide logistics planning for military invasions rather than civilian evacuations.

This scenario requires secure system communications. The data must not be compromised. What would happen if recommendations on military tactics were intercepted? The data must

be correct. What is the impact if the satellite data was modified while being transported between the database machine and the graphics workstation?

The THETA system could provide the secure, heterogeneous interoperability needed in the above example. THETA encrypts network communications to protect data confidentiality. THETA provides data replication to improve data integrity and availability.

1.2.2 Technical Hurdles

Moving from stand-alone systems to networks of distributed hosts introduces new problems due to physical separation and heterogeneity. Some of the technical problems that have to be resolved are:

- Naming and identification of network resources. For processes to reference a resource in an access request, there must be some kind of common naming policy among the network hosts.
- Protection and access control of resources. Permitting remote access of resources now requires remote authentication and authorization.
- Data translation issues resulting from heterogeneous data encodings and data representations. Sharing data across different hardware architectures surfaces a new need for a translation layer between common external data representations used in network messages and local internal representations of data. For example, some machines organize bytes differently, some use "most-significant-byte first" and the others use "least-significant-byte first"; without a translation layer, all data transferred between these hosts would be garbage. Another more abstract example would be if two systems have different representations of a data file; a translation layer can provide the necessary mutations to allow sharing the file object between the two hosts.
- Message based interprocess communication (IPC). IPC for remote processes needs to be message based since memory sharing is not practical between distributed components in most heterogeneous systems.
- Service and resource structures need to be remodelled for remote accesses. Interfaces for important resources need to be clearly defined and may require redesign.
- Errors and error recovery. Problems with distributed accesses are more difficult to discern since there are more sources of errors and more components to the system. Also, error recovery actions may not be so clear.
- Maintaining data consistency among multiple copies of data. To improve data availability and survivability, it is common practice to keep a few copies of data on different hosts in case one host is unavailable at a crucial moment; however, it becomes more difficult to synchronize data modifications among all of the copies.

- Synchronization and control problems. The state of the system is distributed and locking up each host in order to get an accurate snap shot of the system is not always feasible.
- Accounting and administration issues. In a network, each host is controlled by a different administrator. Each of these administrators must cooperate for successful interoperability.
- Verification, debugging, and performance measurement. These issues are more complex.

Despite this long list of concerns, it is not complete. Though it appears daunting, these problems are mitigated through the use of layering, abstract objects, and message passing.

1.2.3 Layering and Abstractions Ease the Burden

Distributed computing is difficult. Dealing with the details of every different system can be overwhelming. As complexity of a computer system increases, the necessity for abstractions also increases. A modular, layered design removes the details of a particular system's internal structures, mechanisms, encodings, and algorithms.

Complex systems can be broken down into logical units that are more easily understood. Each unit needs to be clearly defined in terms of the resources that it controls and the interfaces that allow access to those resources. This modular approach has the following advantages:

- alternate implementations of a unit can coexist,
- unit modifications are less noticeable to entities requesting resources (unless the changes were made to the unit's interfaces), and
- verification, debugging, and testing is much easier on a per-unit basis than on a large, complex system.

Abstractions that define information in terms of data characteristics and methods of accessing that data are often called *objects*. In the following chapter, we define and describe objects and object models. In Chapter 1.4, we describe THETA's layered architecture that implements its object model.

1.3 Object Model

Distributed systems are hard to design and build. Support for high-level abstractions helps disperse the difficulties inherent in distributed programming. An object oriented paradigm hides the internal complexities and differences of network resources and provides a uniform interface to all data objects in the system.

An object model defines all activities in terms of *accesses* on *objects*. For example, when logging in to a system, the user is not interacting with the login daemon; rather, the user is performing the operation "login" on the object "console". Or perhaps, embellishing on the concept of abstraction, a user could be considered to be performing the action "open" (that is, login) on a "door" (the user's account) to a "room" (the console) using a "key" (the user's password). Object models exploit a powerful metaphor to make systems easier to conceptualize.

1.3.1 What Is an Object?

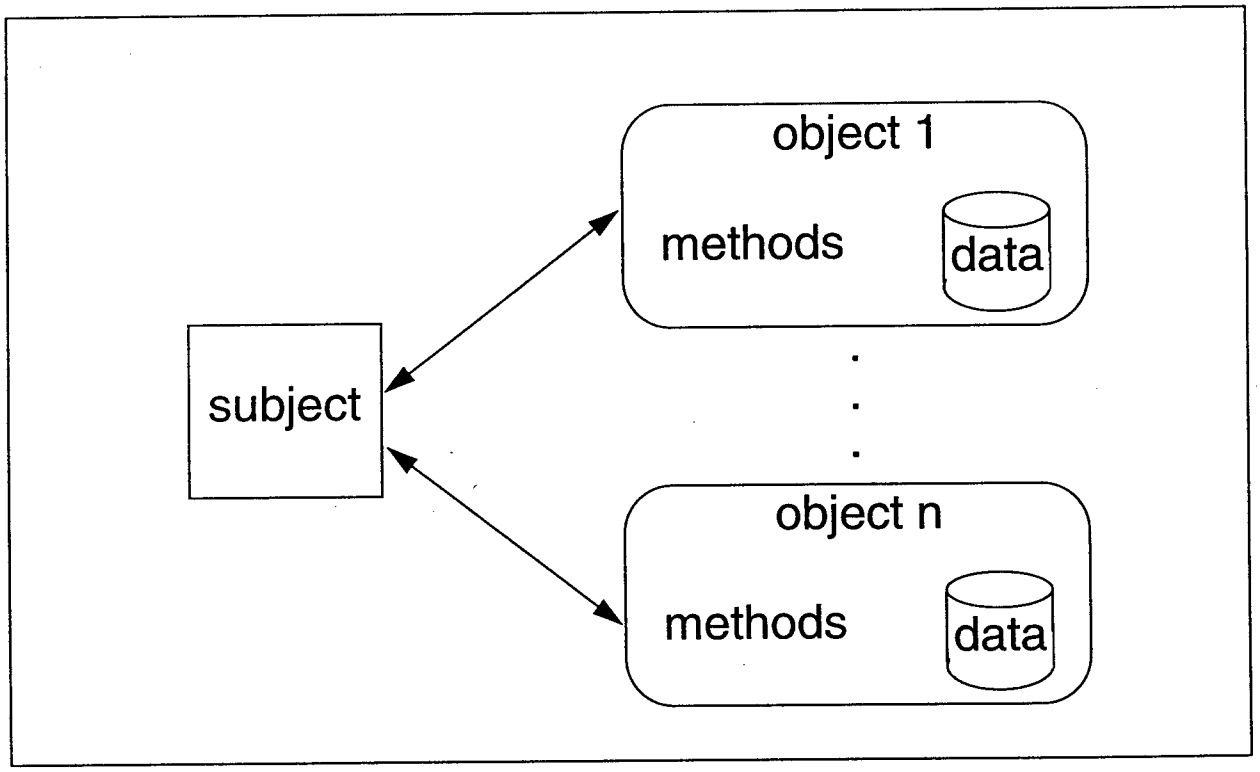

Figure 1-2: THETA Object Model

Objects are abstractions of resources like processors, memory, and devices. An object is a combination of data and methods used to access that data. Figure 1-2 depicts a subject, which in THETA is a principal or group, accessing an object's data through the methods defined for

the object. We refer to methods of manipulating an object as *operations*. Subjects cannot do anything to an object unless the action is through a defined operation.

Objects can be accessed only by invoking operations on them. THETA users (that is, principals) start client programs to issue such invocations. Operations are implemented by object managers. A manager hides the internal representation of the objects it manages, and provides a precisely defined interface to these objects. Some or all of the internal representations of a manager's objects are stored in an object database (ODB).

1.3.2 Object Types

All THETA resources are organized into groups of objects that have similar characteristics. All objects with the same characteristics are said to be of the same *object type*. When an object type (or simply, a type) is defined, the developer specifies the common characteristics of a type, which include the data structures in the object, the operations that can be performed on objects of this type, and the rights needed to invoke the operations successfully. Figure 1-3 shows a sample operation definition from the *Set* type specification file. In the example, we define the operation *ShowSet* to have the input parameter *SetName* and the return data *SetCont*. The access control checks are also stated. The MAC check is "*mac test read*", which means the subject invoking the operation must dominate the level of the object being accessed. The DAC check is specified in the line "*requires view*". This requirement states that

```
generic operation ShowSet mac test read
(SetName:ASC;)
returns
(SetCont:SET_CONTENTS;)
requires view;
```

Figure 1-3: Example Operation Signature from the *Set* Object Specification

the invoking subject must have the *view* right on the *Set* generic object[1] to successfully perform the operation *ShowSet*. For more information on type specifications and operation declarations, see the *Manager Developer Tutorial for THETA, Volumes I and II* [32].

1.3.2.1 Subtypes and Inheritance

THETA types are hierarchical. Each type, with the exception of root type, *Object*, has exactly one parent. An object type inherits the attributes defined for the parent type and all other ancestral types. As shown in Figure 1-4, all types inherit the data attributes and operations that are defined for the ancestor type *Object*. Notice that *Directory* objects also inherit attributes

[1] Generic objects are explained more in Section 1.3.2.4.

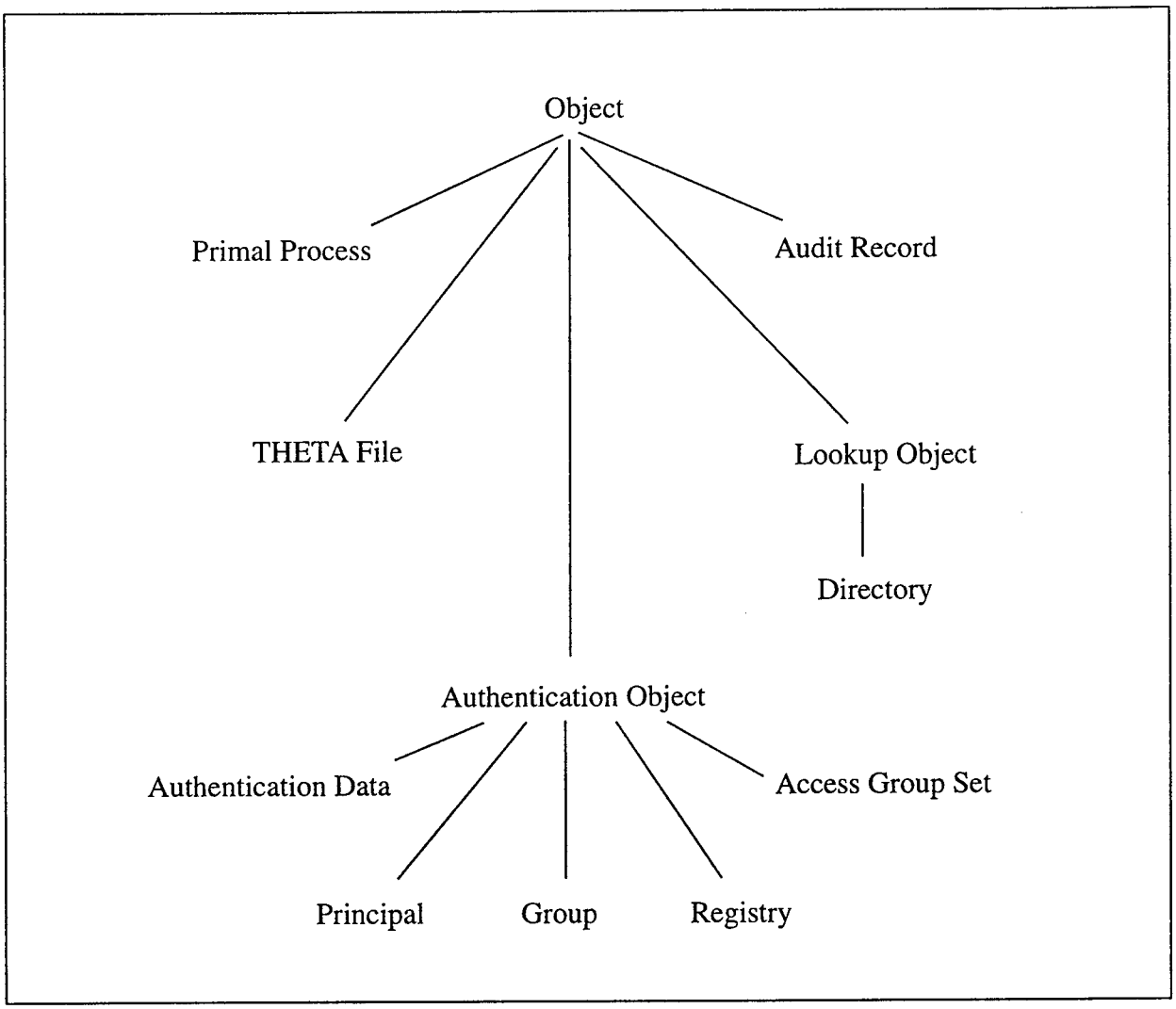

Figure 1-4: Portion of the THETA Type Hierarchy

from the *Lookup Object* type as well.The *Lookup Object* type is a *subtype* of *Object*, and *Directory* is a *subtype* of *Lookup Object.*

Inheritance is the mechanism used to share type descriptions among several distinct types that have some similar attributes. The common attributes are defined in the parent type, or some other ancestor type. Operations defined for ancestor types do not need to be rewritten for sub-types; thus, inheritance promotes code-reuse.

1.3.2.2 Styles of Types

Each manager of an object type has a strategy for maintaining the databases. This strategy depends on the *style* of the type. In THETA, there are four type styles:

- ancestral - Ancestor types allow common data structures, operations, and other object attributes to be declared in one type specification file, yet these types can be imported and used by other types. See Section 1.3.2.1 for a description of *inheritance* and *hierarchies*.
- primal - Primal types are types whose objects have meaning only on the system where they were created. For example, a memory address on one machine does not refer to the same information as a memory address on some other machine; thus, memory addresses are *primal*. *Primal* objects must remain on the machine where they were created because that is the only machine where the data makes sense.
- distributed - Distributed types can have objects that reside on several hosts; however, the manager of the type does not enforce any data consistency rules. For example, say we have a distributed type *File*. If we copy a file object named *Original* from host A to host B, and then edit the copy on host B, the two files are no longer identical, yet they have the same name.
- replicated - A replicated type has objects that have *copies* of the object image on several hosts. When a replicated object is created or modified, all copies are updated so that the collective object database across the network remains consistent.

1.3.2.3 Object Instances

Each object that is a member of a type is called an *instance* of that type. Operations can be performed on specific objects or the entire class of objects. A client must make it clear which object is the target of the operation. To distinguish the target when invoking an operation, the client must supply the identity of the exact object instance that is to be accessed. Object identification is discussed further in Section 1.3.3.

1.3.2.4 Generic Objects and Generic Operations

A generic object is a single object that represents the class of objects of a type. The kinds of operations that are usually declared for the generic object of a type are create, delete, and search. In THETA, there is only one generic object per type per security level. The example shown in Figure 1-3 defines a search operation on the *Set* generic object.

1.3.3 Object Identification

THETA provides a global and location-transparent way to identify objects. By global, we mean an object name can be issued from any location to uniquely identify an object anywhere on the network. By location-transparent, we mean an object's location is not encoded directly in the object name.

There are two kinds of names for THETA objects: unique identifiers and *Directory* objects. The Unique Identifier (UID) is a machine-generated name; the *Directory* object is a user-selected symbolic name. Each THETA object has a single UID, which is stored with the object and is bound to the object at object creation time. A sample UID may look like *{192.76.175.200:4:9:#UNCLASSIFIED:Principal}*. The UID's awkward external representation is due to optimizations made for internal handling by the machine.

Identifying objects via their UIDs is not very intuitive for most users. Users typically want to refer to objects using symbolic strings that are meaningful to them. The Directory Manager provides a distributed and replicated service that maintains a mapping between user-defined symbolic names and system-maintained UIDs. The Directory type provides a hierarchical naming structure. An example of a symbolic name is**: a:b:c** where **a** and **b** are subdirectories and **c** is the object being referenced. Directories in this path are non-decreasing in security level. See the *Software Design Document* (SDD) for the Directory Manager [34] for more details on using a Directory object to reference another object.

A THETA object is not required to have a symbolic name. An object may have no, one, or more then one symbolic name. If there is no symbolic name for an object, the object must be accessed using its UID.

1.3.4 Object Replication

THETA provides reliability and availability by supporting replication of objects at multiple sites. A *replicated* object is one for which more than one copy is being maintained, and the replicas reside on more than one host. Each replica of the object has the same UID. The object may be accessed through a manager on any of the hosts where it resides.

Data consistency of replicated objects is maintained by the version vector scheme (see [22] in "References" on page 91). The classic problems of availability and consistency are resolved by allowing read and write quorums to be set for each replicated type at type definition time.

1.4 THETA Architecture

THETA uses a layered architecture to implement the object model, which is discussed in Chapter 1.3. The THETA clients, managers, and kernel are implemented on top of an existing secure Constituent Operating System (COS). THETA has two major design goals with respect to COS systems. First, THETA must be incorporated into the host system without modifications to the COS. Second, THETA must be able to operate on a network of heterogeneous hosts.

A COS must meet TCSEC B3 security and assurance requirements for the THETA system to be B3. The following features are expected of the COS:

- Assured process separation - MAC, DAC, and user and process identification mechanisms of the COS control all direct interprocess communication (IPC). No IPC mechanisms are permitted to override these checks.
- Non-interference with process operation - No untrusted process is permitted to interfere with THETA processes responsible for security. As stated above, the COS mechanisms that enforce non-interference are MAC, DAC and user and process identification.
- Stable storage - The COS file system, file permissions, and COS MAC labels protect the data needed for enforcing security and for maintaining object databases.
- IPC support - THETA IPC primitives and protocols rely on trusted path, local IPC, and TCP/IP facilities of the COS.
- Device support - COS device drivers are used for device support.

1.4.1 THETA Components

The THETA object model is implemented in layers. A simplified view of the hardware and software components of the THETA system is depicted in Figure 1-5. The major pieces are

- Constituent Operating System
- Network software and hardware
- THETA kernel
- THETA managers
- Objects
- THETA clients

In the diagram, the THETA kernel does not completely obscure access to the COS. The picture is drawn this way on purpose to stress the fact that THETA managers, clients, and users can still interact directly with the COS if desired.

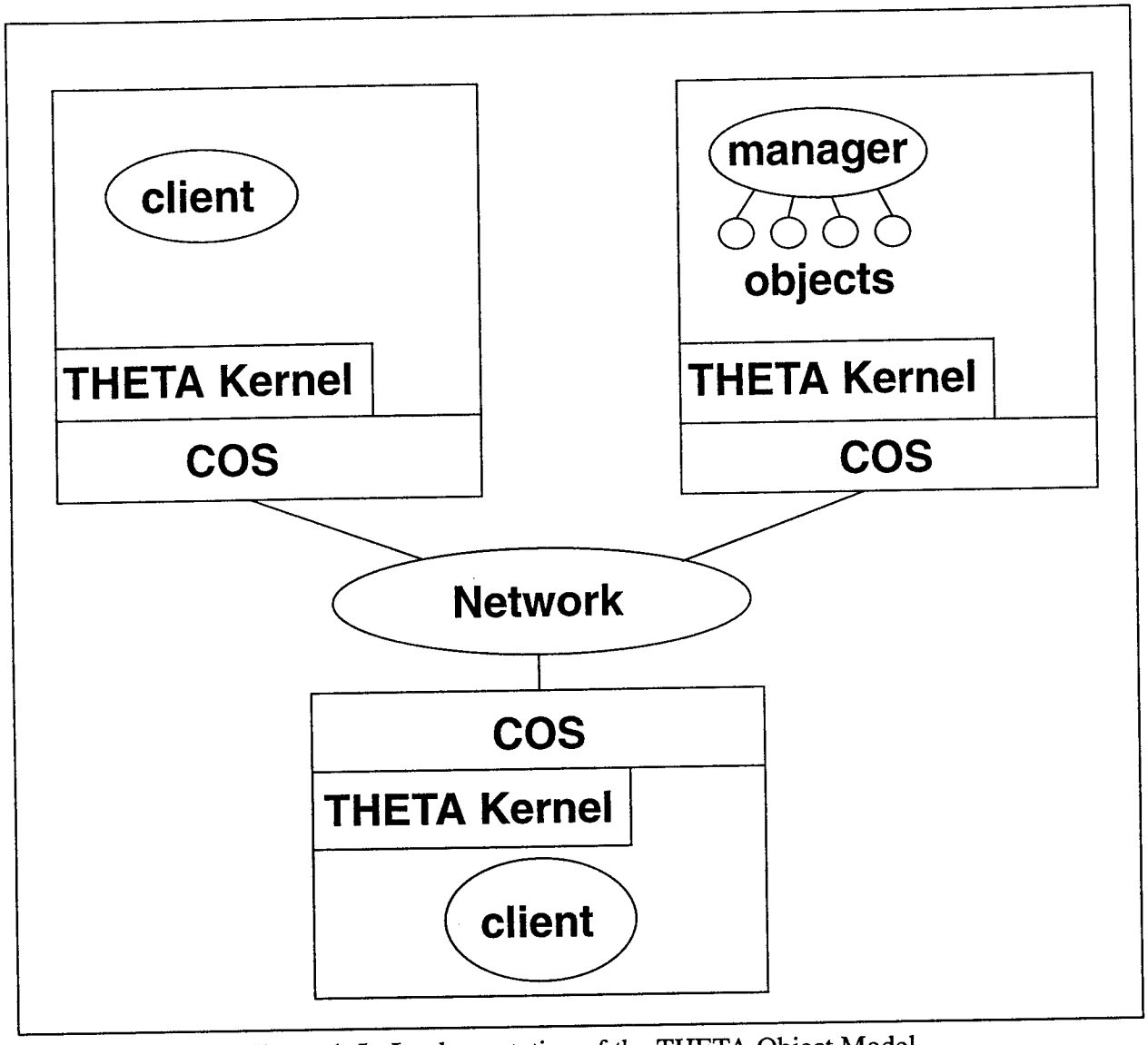

Figure 1-5: Implementation of the THETA Object Model

1.4.1.1 Constituent Operating System

THETA provides abstractions of system resources for higher level applications. These abstractions must eventually map to some real system resource like processors, storage, and devices. The COS is the true controller of these real system resources. THETA interacts with the COS to manipulate the THETA objects (which are abstract views of real system resources) in the manner declared in the object type specifications. THETA relies on the system calls of the COS to be implemented correctly and for the COS security mechanisms (MAC, DAC, privilege schemes, etc.) to be enforced on those system calls.

Consult the vendor-supplied documentation for more information about the security mechanisms, their implementation, and the assurance arguments for the COSs on a THETA network.

1.4.1.2 Network Software and Hardware

For THETA to be a distributed system, it must operate over a network. THETA relies on the network hardware of each machine (the Ethernet cards, the cables, the routers, etc.) and on the software and configuration files that implement the protocol.

As a networked service, rogue processes may snoop data transmissions that are sent over the network. To combat this threat, THETA implements its own packet encryption of messages. When secure networks and secure protocols become more stabilized and more readily available, we will remove encryption from the THETA processes and rely on secure COS network services.

1.4.1.3 THETA Kernel

The THETA kernel is responsible for authenticating COS users and their associated processes, registering those processes, enforcing MAC checks on IPC messages and replies, supplying location-transparent access to objects, message forwarding, and message upgrading.

Currently, there are two different implementations of the THETA kernel. Their architectures are different enough that it is beyond the scope of this introduction to discuss them here. Consult the *Software Design Document* (SDD) for the THETA Kernel [34] for details on the two architectures, their components, and their interactions.

1.4.1.4 THETA Managers

For each object type, there is a THETA manager that regulates accesses to the object instances of the type. When clients invoke operation requests on objects that are regulated by the manager, the manager first verifies that the request passes all MAC and DAC checks. If the request is valid, the manager accesses the object database that is maintained in secondary storage on the local host. The THETA manager relies on COS mechanisms like file ownership, group ownership, file permissions, and MAC labels to protect the object database from being accessed outside of THETA interfaces.

When a manager is created, a *Program Support Library* (PSL) of operation function calls is generated. This library contains the code that client programs should use to invoke operations on objects of a given type.

For details on the components and workings of the THETA managers, consult the *Software Design Document for THETA* [34].

1.4.1.5 Objects

As stated in the previous chapter, an object contains data *and* the methods used to access that data. In THETA, we often refer to the data portion of the object *as* the object itself. THETA objects are kept in object databases (ODBs) on COS file systems. COS and THETA mechanisms protect the data from improper accesses.

1.4.1.6 THETA Clients

THETA clients use the common interface protocols defined in various PSLs to invoke operations on objects. A client can use the services of several THETA managers that reside on several nodes on the network without knowing the details of object location or data representation. The managers and kernels mask the complexities of the object away from the client and user so that the system is easier to program and easier to use.

1.4.2 THETA Communications

In a distributed, object-oriented system, client processes access information by sending requests to manager processes. To the client, data accesses are simple. The THETA client needs to get input, prepare the THETA operation, invoke the operation via the PSL routines, wait for the response, and process the output. The only THETA-specific step is the PSL call that provides location-transparent access to the requested objects. PSL is a synchronous communication request; however, the client program can be written to use the lower communication levels within the PSL to achieve asynchronous processing.

Figure 1-6 shows the path of the PSL call through the THETA system. The diagram is very simple. The first step of the PSL call is to register with the THETA kernel. The kernel verifies the identity of the user who is running the process. If the user is known in the THETA configuration, and the MAC level of the invocation is within the range of the user, registration succeeds. After successful registration, the PSL sends the invocation request on to the kernel; part of the request must indicate the UID of the target object, which also specifies the type of the object. To process the request, the kernel must determine which manager controls objects of this type and where a manager (at the chosen MAC level) is running on the network. The kernel sends out a *locate* request to the other kernels on the network to find a manager service at the appropriate level. Each kernel checks to see if it can service the *locate* request. If it can, it responds to the original kernel with a *found* message. The original kernel then sends the PSL call on to the kernel that had the service available. The remote kernel passes the request on to the manager. The manager performs any DAC checks that may have been specified for the operation. If the user that invoked the request from the remote client has the necessary privileges, the manager processes the request and sends a reply back to the kernel. This kernel forwards the reply to the original kernel who then sends the information back to the client.

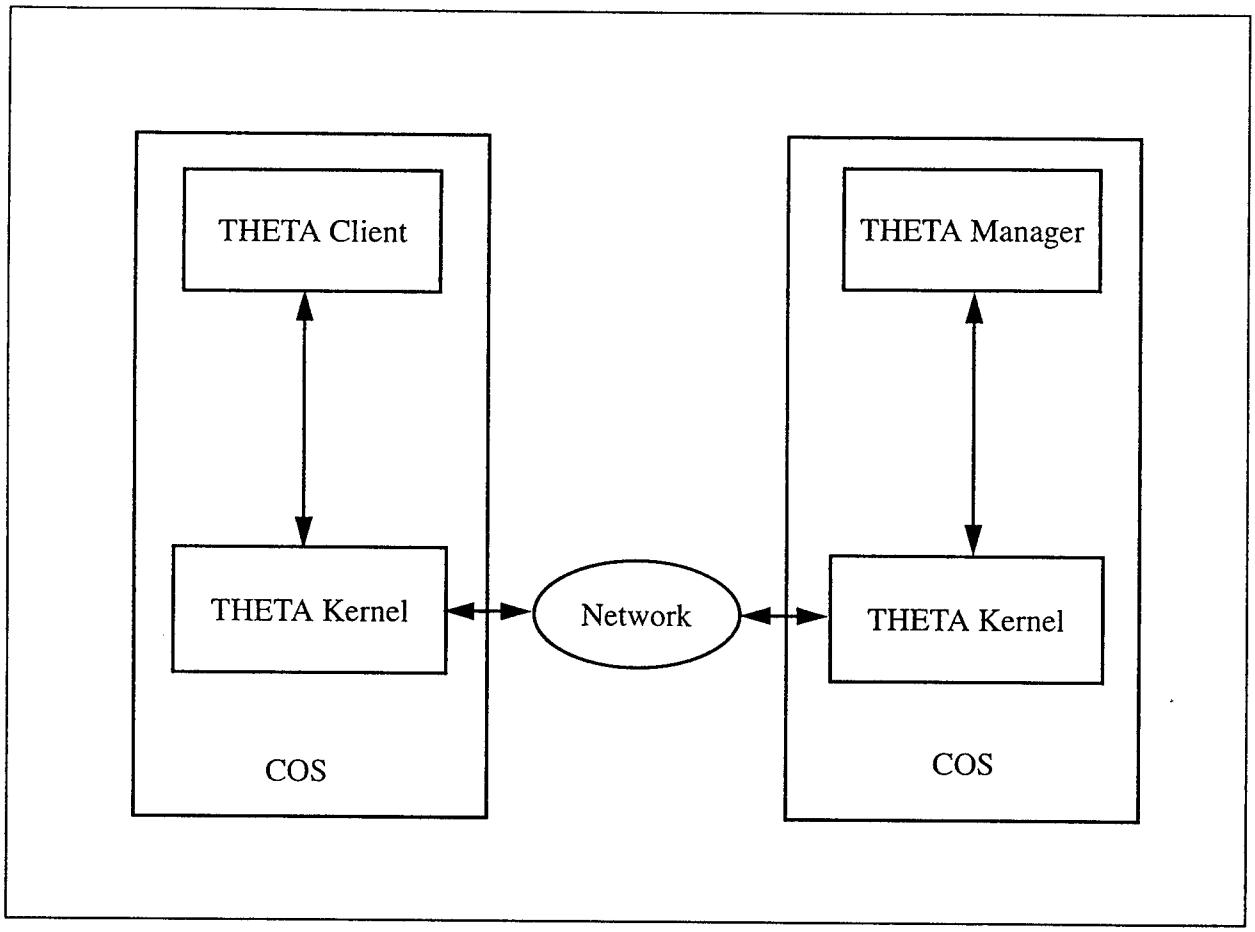

Figure 1-6: THETA Communication Paths

1.5 Security Policy

The THETA security policy is stated in the *Software Requirements Specification for THETA* [35] and formally addressed in the *Formal Security Policy Model for THETA* [31]. However, we briefly cover some important aspects here. The security policy for THETA adheres to the specifications described by the TCSEC for level B3 systems. The policy can be categorized into the following:

- A discretionary access control (DAC) policy that is designed to restrict operations on data objects according to the identity and privileges of the client.
- A mandatory access control (MAC) policy that controls the flow of information according to the security levels of client and object.
- Some additional policy rules that define the security configuration, guarantee of trusted paths, etc.

The MAC and DAC policies are clearly separated. In fact, they operate at different granularities in the object model. The mandatory policy is enforced largely at the message passing level. The discretionary policy is enforced at the object level. The MAC and DAC policies state global constraints for the entire system rather than for individual hosts.

In all secure systems, the amount of code that performs trusted operations should be kept to a minimum. In THETA, the kernel performs trusted operations. All THETA system managers contain trusted code. In general, any MLS manager has trusted code regardless of whether it is provided as part of THETA or by the user. The reason for this is that the ACG automatically generates trusted code so that the manager can be run at single level or MLS. In THETA, the system relies on the constituent operating system (COS) security mechanisms whenever possible to enforce the security policy rather than duplicating code.

1.5.1 Mandatory Access Control

A distributed operating system mandatory policy must be defined in terms of message passing between active entities, rather than the traditional Bell and LaPadula read and write operations of an active entity on a passive entity. In THETA's object-oriented paradigm, data is accessed through a well-defined set of methods. When a subject wishes to perform some operation on an object, the access request is sent as a message from a client process (acting on behalf of the subject) to a manager process. So, the object is not a passive entity; it can be accessed only by making regulated requests to the managing process, that is, via messages.

The THETA MAC policy has two components:

- Rules for message passing to prevent direct downgrade of information.
- A policy for multilevel entities to prevent compromise of information via covert channels.

The multilevel security policy is based on a theory of information flow security developed at ORA. This theory is detailed in [13] and [11]; however, we provide a brief description below. The policy defines information flow in terms of deductions that can be made about unseen (higher security level) events in a system's history. This policy is called "restrictiveness". The restrictiveness policy defines security as follows: a system component is secure provided it does not allow information to flow from high security levels to lower ones.

1.5.1.1 MAC Labels of Subjects and Objects

In order to enforce the mandatory access control policy, the THETA system must compare the MAC label of the invoking process with the MAC label of the target object. The MAC tests are dependent on the type of operation. A process may view an object that is at or below its level, and a process may write to an object that is at or above its level. MAC labels of processes and objects are set at the time they become "known" to THETA.

When a process registers with the THETA kernel, the process is "stamped" with the identity and security level of the user who started the application. If the process is started over a range of levels, there is one "currently active level" that is within the range and all invocations are marked with the MAC label of this active level.

A THETA object is created when a successful *create* invocation is processed by the manager. The *create* invocation is a message, which comes into the manager at a single MAC level from a registered THETA process. The manager creates the object at the MAC level of the *create* message. The MAC level becomes part of the identifier of the object, and the object is fixed at that level for the duration of its existence.

1.5.2 Discretionary Access Control

Since object managers are the entities that implement operations on objects and DAC restricts operation executions, all THETA managers enforce discretionary access control on their objects. As a part of each object, an access control list (ACL) is maintained to indicate which users may perform what operations on that object. The DAC policy is necessarily object-dependent since operations and their semantics vary according to their type.

1.5.2.1 Access Control Lists

As stated above, every object has an access control list. An ACL is a list of subjects[1] and their corresponding access rights to the object. Just because a subject has access rights on an object does not mean that the subject can perform any arbitrary action on the object. Accesses are restricted as part of the definition of a type. Every object is a member of some type, and as part

[1] In THETA, subjects are principals and groups.

of the definition of a type, the operations that can be performed and the rights required to invoke the operation are specified. Therefore, a subject that is attempting to access an object using a particular method must have the access right required to perform that operation.

To clarify, for each operation, the required privilege is specified as part of the *signature* of the operation. An ACL contains a list of subjects and their access rights for a single object. When a subject invokes an operation, the operation signature is consulted to determine what access rights are needed, and then the ACL of the object is checked to see if this particular subject has the necessary privileges.

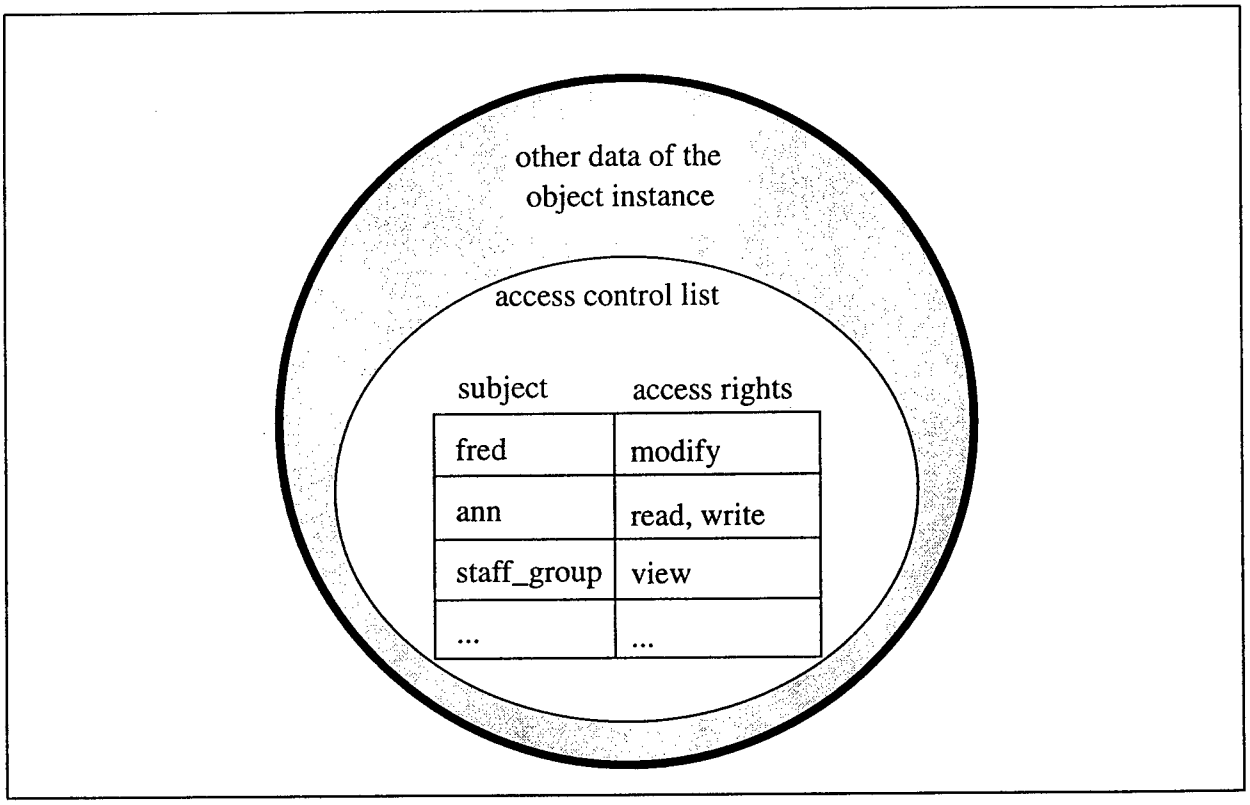

Figure 1-7: An Access Control List within an Object Instance

1.5.2.2 Access Group Sets

An access group set (AGS) is a principal and a list of groups that have that principal as a member. The principal (and any processes that the principal starts) inherits the access rights of all groups specified in the AGS.

1.5.2.3 Process Bindings

When a principal starts a process that registers with the THETA kernel, the process is bound with the AGS of the principal. By *bound*, we mean that the access rights of the process are restricted or *bound* by the value of the AGS.

1.5.3 Other Forms of Access Control

More restrictive access control schemes like "unique operator"[2] requirements can be enforced in the operation code within a manager. Security checks in the operation code must be added by the developer. Note that the security checks can become *more restrictive* only by adding code to the manager operations. THETA MAC and DAC checks cannot be overridden.

1.5.4 Encryption

THETA operates over a network and is therefore subject to the standard risks of networking. One of the main threats operating over a network is wire-tapping. To combat this risk, THETA provides encryption of datagrams that are sent over the net. For more information about the trusted networking capabilities, see [39], [40], and [41].

1.5.5 Assurance Arguments

THETA has a layered trusted computing base (TCB) consisting of a message TCB and an object TCB. The TCB size depends on how the administrator configures the system. There is a trade-off of assurance versus trusted application flexibility. To meet B3 criteria, only the message TCB can run multilevel.

Despite THETA's long existence, there are still no standards for evaluating secure, distributed operating systems. We have had to interpret the TCSEC and the TNI to forge our own criteria. As a result, THETA's approach to security engineering has combined traditional, conservative methods with more liberal, experimental practices. We have tried for the best of both worlds, and we have had considerable success.

The THETA kernel development emphasized the traditional approach. It implements the basic functions of the Cronus kernel, but is completely redesigned and reimplemented. The code that is trusted is minimized. Sizes of the various kernel components are detailed in the *Computer Systems Operator's Manual for THETA* [29].

---

[2] "Unique operator" requirements come up in various separation of duty scenarios. For example, in a banking scenario, a check may need to be signed by two unique supervisors before it can be cashed.

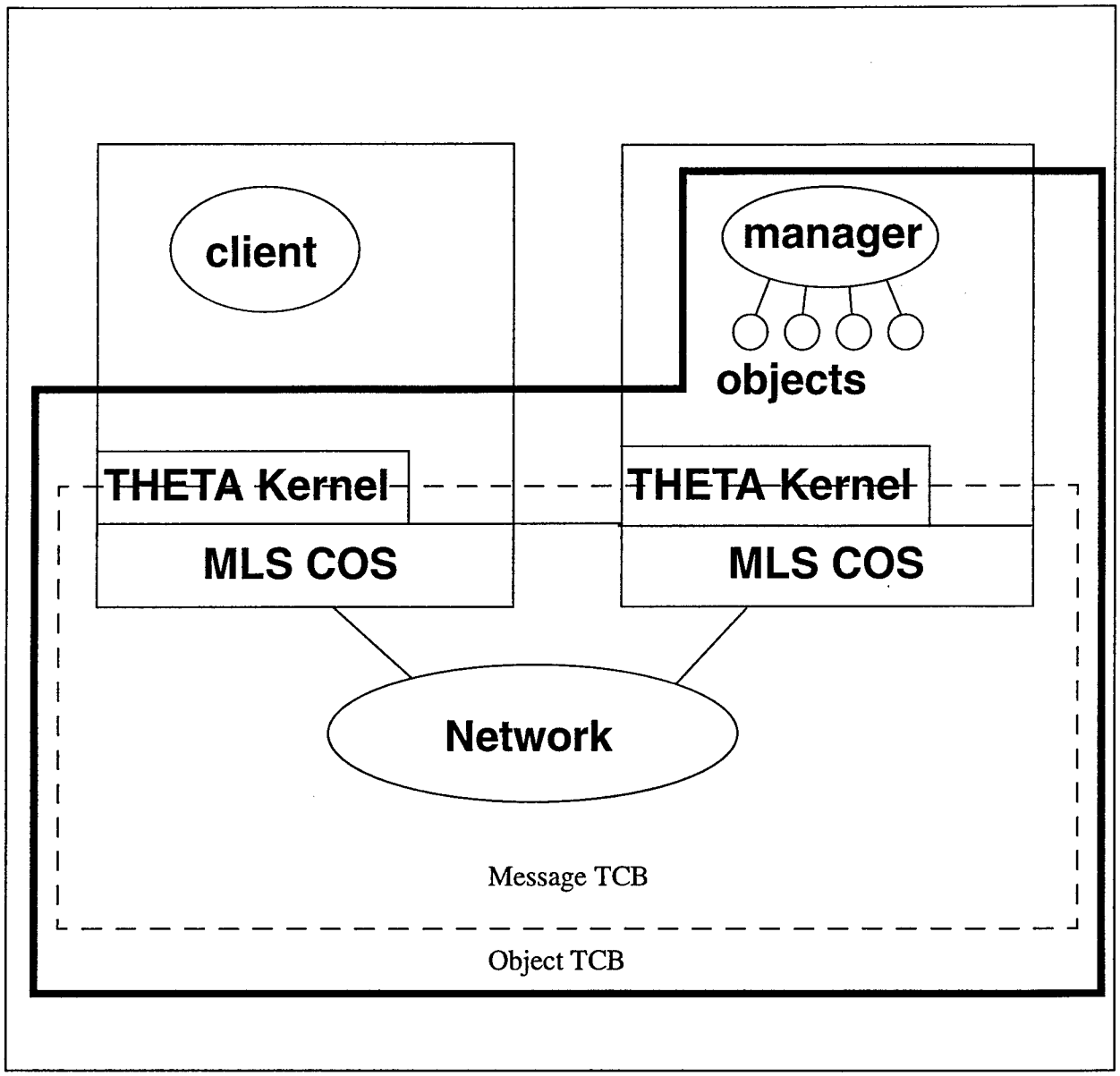

Figure 1-8: THETA Trusted Computing Base Boundaries

The THETA manager development focuses on *trusted extensibility*, which is the idea that THETA may be developed and adapted to new applications by adding new trusted software. Several managers, trusted and untrusted, have been autogenerated for the fielded THETA system. New managers may be created by users and installed in THETA at any time.

The key THETA advance is the structuring of the manager autogeneration process to simplify the arguments that must be made for security assurance of new trusted managers. Trusted extensibility is discussed more in Section 1.5.6 below.

Running a manager as a multilevel process does increase the amount of trusted code, which may not always be acceptable. THETA managers are designed so they can be run either as untrusted, single level managers or as trusted multilevel managers. Changing from one to the other is a mere matter of getting the THETA operator and THETA administrator to shut the system down, adjust a configuration file, and re-boot. Despite this simple procedure, managers should not be run across multiple levels unless the code has been carefully written, inspected, and rigorously tested.

THETA is actually two systems in one:

- the *trusted* THETA, which includes the kernel and the suite of MLS managers;
- the *development* THETA, which includes untrusted autogeneration tools for creating new MLS and MSL managers and an untrusted universal application interface.

The development tools are not themselves trusted, because the trusted managers they produce are subject to manual inspection and testing before delivery. However, there is the possibility of having trusted manager development tools in the future.

Assurance of THETA's multilevel security is based on a formal security policy called *restrictiveness*. Restrictiveness limits the ways in which information can flow within a system, and when applied to multilevel security, it prohibits information flow from high security levels to lower ones. In the restrictiveness policy, *information flow* is defined in terms of facts that can be deduced based on observations of a system's behavior.

THETA is unique in its use of restrictiveness in its security policy. This choice of policy has several advantages.

- An information flow policy is superior to access control policies (e.g., Bell-LaPadula) in that it provides a framework for discovering covert channels.
- Restrictiveness is superior to most information flow policies in that it offers *composability*: communicating restrictive components form a restrictive system. This property is essential for security analysis of distributed systems.

Two restrictive (that is, secure) subsystems can be hooked together to form a larger restrictive system. The THETA policy requires the THETA TCB to be restrictive. Since restrictiveness is a composable property, it is sufficient to demonstrate that the components of the TCB are restrictive. The fact that security verification can be decomposed in this fashion is a tremendous advantage when trying to build a distributed secure system such as THETA. Composability can also be exploited to add multilevel services and hosts to a distributed system in a secure manner without the need for re-verification of the entire system.

We used Romulus (a tool developed at ORA) to formally model and prove the restrictiveness security properties of the THETA kernel. Previously, ORA developed techniques for demonstrating compliance with restrictiveness using the Gypsy Verification Environment (see [42]).

1.5.6 Secure Extensions

As new types are defined and new managers and clients are built, the THETA system is extended. It is important that extensibility be a simple exercise that does not compromise security.

The mandatory policy constraint on information flow is that the THETA system be restrictive. Processes outside the TCB are at a single-level and therefore are trivially restrictive. Because restrictiveness is composable, it is sufficient to show that every component of the TCB is restrictive.

The key problem for secure extensibility is guaranteeing that new MLS managers added to the system are restrictive. Managers are usually complex software. It is legitimate to ask: why should any manager be part of the TCB? This question has been considered in [24]. The advantages of having some MLS managers are increased efficiency and greater functionality. The efficiency is gained at the cost of assuring that software added with new managers enforces the THETA security policy.

A large portion of a manager's functionality is independent of the types managed. So, a significant amount of a manager's design and implementation is invariant over types and can be reused. If a manager was generated using the ACG tools, only the type-specific functionality would need to be supplied by the manager programmer, and thus, only that newly supplied code would need to be verified. For example, the security and audit checks required for specific manager operations are autogenerated. The assurance of security is then divided between the manager generation tool, which is a one-time assurance effort, and the manager operations, whose assurance must be determined on a manager by manager basis.

1.6 Object Managers

An object manager is a process that controls a database of objects of a given type (or types) and regulates the accesses made on objects of that type. Clients access objects by making requests to the manager of the object type, and the manager then honors or denies the request depending on the access rights of the client and the rights required to perform the operation.

1.6.1 Tools for Generating Managers

THETA provides the programmer with a set of tools for manager generation. The programmer creates a type specification and a manager specification. The specification files are then processed by the manager generation tools. Successful processing produces several files of code that make up an object manager skeleton. The skeleton implements message packing and unpacking, conversion from canonical to internal representations of data and vice-versa, mandatory and discretionary access checks that may be necessary for an operation, and many other routines common to most managers. To finish implementing a manager, the programmer fills in code for the specific operations that the manager supports.

These tools have been used to generate all THETA managers.

For extensive detail about the specification grammars, the autogeneration process, and the generated code, see the *Manager Developer Tutorial for THETA, Volumes I and II* [32].

1.6.2 System Managers versus Application Managers

Managers are often classified according to the type of object managed. THETA defines a small number of types likely to be of importance to every application. These types are sometimes called *system* types, to distinguish them from other types that can be added by THETA developers. System managers control system types and application managers control other non-essential types. There is no fundamental difference between system and application managers; the distinction is in the role they play in the system rather than any implementation difference. More specifically, application managers may assume the existence of system managers.

1.6.3 Security Range Options

A manager may run in a single level or across a range of security levels. A single-level manager manages objects only at its security level and is implemented as a single-level process. A multilevel object manager can handle operations over a range of security levels. A multilevel manager may be designed as a single multilevel secure (MLS) manager process or multiple single-level (MSL) manager processes. If it is implemented as a MLS process, then the manager is part of the mandatory TCB and is trusted to perform mandatory access checks.

When a service is needed across levels, should the manager be multilevel or multiple single level? There is a fundamental difference between the two approaches. The MLS manager is trusted, and it can, therefore, enforce a security policy different from that of the constituent operating system. The MSL managers, on the other hand, are bound by the COS's security policy.

When installing a manager service into the THETA system, the THETA administrator and site security officer must assess the functionality desired against the trade off of increasing the TCB size. Installing a manager to run across many security levels as a single process increases the size of the TCB; however, installing a manager to run at several single levels limits the capabilities of the manager and uses more system resources. With the introduction of each new service, the system administrators must weigh the consequences of each type of installation and decide which is best on a per-manager basis.

Because objects of the system types will most likely be used by clients at all levels, THETA system managers are implemented as multilevel services. As a part of the TCB, these MLS managers must undergo the necessary certification procedures.

1.6.4 System Managers

Below, we briefly highlight the features of the current THETA system managers.

1.6.4.1 Audit Manager

The Audit Manager collects information about the actions of all THETA processes, including the managers, clients, and the kernel. Audit data is used to detect and locate attempts to circumvent the THETA security mechanisms.

Every secure system starts with the intention to prevent security compromises by imposing a sufficiently strict security policy and by implementing that policy correctly. However, for practical reasons, this intention is not always realized. The security policy may be inaccurate or inadequate for reasons not originally foreseen. Programmer oversight may cause the policy not to be implemented exactly. Small compromises (e.g., covert channels) may be tolerated to improve performance or to give more functionality. Whatever the reason, sometimes system security is breached; therefore, it is wise to have an audit trail to track attempts to penetrate the system. Penetration attempts typically rely on unusual circumstances and actions to defeat a system's security mechanisms; the attempt can be detected if these circumstances and actions are noticed.

THETA is a distributed system, and attempts to defeat its security may involve actions at many different locations. To detect a penetration attempt, it is best to collect these distributed events at a single location where they can be analyzed as a complete, synchronous list of events. The THETA Audit Manager is responsible for maintaining the audit repository.

What actions are deemed "security relevant" and require auditing? Each THETA manager must decide which of its operations are security-relevant, for which values of its arguments, and for which kinds of reply. The level of auditing is determined by the THETA administrator when a manager is installed. However, this scheme is very static and doesn't allow for any trouble shooting during runtime, i.e., if the administrator notices some activity warranting further investigation, he has to reinstall to get a finer grain or change any of the audit parameters. Managers send the Audit Manager data as THETA invocations on the generic object of type "Audit".

When are audit events sent? We have made the basic design decision that the Audit Manager will not solicit audit reports, but will depend on the managers to send them accurately and promptly.

For more detail on the specific operations, see the *Software Design Document* (SDD) for the Audit Manager and the Manager Skeleton [34]. The SDD for the Audit Manager describes the data types and operations that are particular to the audit services. The SDD for the Manager Skeleton details data types and operations that are common to all managers.

1.6.4.2 Authentication Manager

The Authentication Manager, also designated authen, manages five types, listed here with their abbreviations: Principal (prin), Group (group), THETA AccessGroupSet (thags), Registry (reg), and Authentication Data (acdb). These types and operations directly or indirectly support THETA discretionary access control policy.

Principals. THETA principals loosely correspond to user names in traditional operating systems. A Principal object is maintained for each legitimate THETA user. When a COS user launches a THETA process, THETA kernel configuration files are consulted as well as other databases, and the COS user is authenticated. Part of the authentication process is to map the identity of the COS user to the identity of a THETA principal.

A principal can never have more than one COS user per host; however, a principal may have no COS user counterpart for a particular host in the network. The THETA system administrator is responsible for maintaining this information. See the *Software Design Document* (SDD) for the Kernel [34] and the *Computer System Operator's Manual for THETA* [29] for details.

Groups. A THETA group is a collection of principals and other groups.

Access Group Sets. An access group set is a set of groups that a principal wants enabled. In other words, even though the principal may be a member of several groups, the user may want membership only in a particular subset to be active for purposes of DAC. A principal's access rights on objects are determined by the active group memberships, that is, the access group set. For an in-depth discussion on DAC checks, see the *Software Design Document* (SDD) for the Manager Skeleton [34].

Registry. The THETA Registry type supplies an easy way to reference principals and groups. There are no objects of this type and the generic objects of type principal and group are used to store the name spaces. This type is used to supply a set of operations that manipulate the principal and group name spaces.

Authentication Data. This data type is maintained for backward compatibility to Cronus; it serves no purpose in the THETA system.

For more detail on the specific operations, see the *Software Design Document* (SDD) for the Authentication Manager and the Manager Skeleton [34]. The SDD for the Authentication Manager describes the data types and operations that are particular to the identification and authentication services. The SDD for the Manager Skeleton details data types and operations that are common to all managers.

1.6.4.3 Automatic Code Generator

The Automatic Code Generator (ACG) Manager (formerly Type Definition Manager) is a tool for developers to generate (possibly multilevel) managers. Programmers write specifications for the object type and the manager that will regulate access to that type, and then process those specifications by making invocations on the ACG. Successful processing of the specification files produces a skeleton of code that the programmer must then tailor by adding code to implement operation semantics. For extensive detail about the specification grammars, the generation process and autogenerated code, reference the *Manager Developer Tutorial for THETA, Volumes I and II* [32].

For more detail on the specific operations, see the *Software Design Document* (SDD) for the Automatic Code Generator Manager and the Manager Skeleton [34]. The SDD for the Automatic Code Generator Manager describes the data types and operations that are particular to the manager generation services. The SDD for the Manager Skeleton details data types and operations that are common to all managers.

1.6.4.4 Configuration Manager

The THETA Configuration Manager is designed to store information about nodes on the THETA network and the manager services that are available on those nodes. The purpose of the Configuration Manager is to provide the THETA administrator a useful tool for managing the network and THETA services.

The Configuration Manager manages three types: *Host_Configuration*, *Service*, and *ActiveServiceList*. Host_Configuration objects contain information about the hosts on the network. Such information would include hostname, Internet address, and the hardware architecture of the host. Service objects contain information about the managers that are installed on the network. Such information would include the manager name, the types managed, and whether the manager can be run across multiple levels. ActiveServiceList objects contain

information about all currently registered managers that are available on the network. This data includes the names of the managers, what hosts they are running on, and at what security levels.

Each network of THETA hosts needs only one Configuration Manager; however, if there is more than one instance, the data must be kept consistent across all platforms. To maintain data coherence, all types managed by the Configuration Manager are replicated.

The Configuration Manager monitors information about the state of the THETA network and keeps that information in a database. This information may be useful to other processes like the Primal Process Manager, in order to perform actions such as automatically starting a manager process at a particular level on a certain host. The THETA Operator may also query the databases to get a view of the services available on the network.

The THETA Operator has another more user-friendly tool available, known as *dream*, that monitors the state of the THETA network. The *dream* application is a graphical application that allows the Operator to monitor and control the THETA processes on the network through "point-and-click" and "drag-and-drop" actions.[1] The information from the *dream* application is not stored in databases, and ceases to exist when the application is exited.

For more detail on the specific operations, see the *Software Design Document* (SDD) for the Configuration Manager and the Manager Skeleton [34]. The SDD for the Configuration Manager describes the data types and operations that are particular to the THETA network configuration services. The SDD for the Manager Skeleton details data types and operations that are common to all managers.

1.6.4.5 Directory Manager

The Directory Manager provides a hierarchical name-space for THETA objects. The name-space is organized into directories and directory entries in a similar fashion to that of the UNIX file system, except that any object, not just files, can be named. A directory object contains entries associating symbolic names with arbitrary object UIDs, but it can also contain entries that associate names with other directory objects known as "subdirectories". All directory objects managed by the Directory Manager are connected together in this fashion into a directory tree with a single "root" directory. The unique full name, or "pathname", of an object then starts with the name of the root directory and includes the names of all the subdirectories that are part of the "path" between the root directory and the entry for the named object. A "relative pathname" starts in some non-root location (the "current" directory) and ends with the name of the object. If the directory entry for the object is contained in the current directory, the relative pathname becomes just the object name itself.

---

[1] The *dream* interface is documented in the *Computer System Operator's Manual for THETA* [29].

An example of a full pathname could be ":usr:theta:object_1". In this pathname, "usr" is the name of a directory, "theta" the name of a subdirectory of "usr", and "object_1" is the name of an arbitrary THETA object that is represented by an entry in directory "theta". The colons, except the first one, are separators used to keep the names in the path distinct. The leading colon, by convention, represents the name of the root directory, ":", so "usr" is actually a subdirectory of the root directory, which can in turn be thought of as a subdirectory of the generic directory object. Directory Manager generic operations operate on the root directory.

For more detail on the specific operations, see the *Software Design Document* (SDD) for the Directory Manager and the Manager Skeleton [34]. The SDD for the Directory Manager describes the data types and operations that are particular to the THETA cataloging services. The SDD for the Manager Skeleton details data types and operations that are common to all managers.

1.6.4.6 Primal Process Manager

The THETA Primal Process Manager is designed to provide information about processes that have registered with the THETA kernel. A successful registration leads to the creation of a primal process object on the host on which the process registered. COS processes are not migratory entities in THETA, and so the primal process objects created to represent them do not migrate either. Primal process objects contain the data that prevents repudiation of actions recorded in the audit log. In particular, the identity of the *COS user* that started a process that caused audit events can be learned only by obtaining this information from this manager.

For more detail on the specific operations, see the *Software Design Document* (SDD) for the Primal Process Manager and the Manager Skeleton [34]. The SDD for the Primal Process Manager describes the data types and operations that are particular to the tracking services for local THETA processes. The SDD for the Manager Skeleton details data types and operations that are common to all managers.

1.6.4.7 THETA File Manager

The THETA File Manager implements a file system for THETA file objects. This manager is responsible for managing objects of type "Theta_File", which implement a distributed file system. The THETA File Manager provides functionality of conventional file systems, like create, read, write, open, close, and remove. Clients can access file objects throughout the THETA system.

File objects are not replicated and stay on the host on which they were created; in the THETA terminology, this type is *primal*. In addition, file objects are stored entirely on a single host. Large file objects cannot be split into pieces and then stored at different sites. (Programmers can, of course, write managers and/or applications that provide this sort functionality.)

For more detail on the specific operations, see the *Software Design Document* (SDD) for the THETA File Manager and the Manager Skeleton [34]. The SDD for the THETA File Manager describes the data types and operations that are particular to the THETA file services. The SDD for the Manager Skeleton details data types and operations that are common to all managers.

1.6.5 Application Managers

The application manager suite consists of managers that provide a user service rather than a system service. The system managers provide services needed by the THETA system to operate correctly and safely. The application managers provide users with their required services and functionality.

1.6.5.1 Account Manager

The Account Manager was created to test role-based access control policies using THETA's flexible access control mechanisms. The scenario is modelled after a "separation of duty" policy that one may find in a bank. Several distinct users must cooperate in order for cash voucher to be honored. The Account Manager controls two types, a persistent object type (account) and a transient object type (voucher). Voucher objects have a particular set of operations that must be performed in a certain order by specific privileged users. The Account Manager uses many THETA mechanisms like MAC, DAC, ACLs, and AGSs to enforce this complicated role-based access control policy. See [23] for a description of the manager and the scenario.

1.6.5.2 Mission Planning and Tracking Managers

Trusted Information Systems developed a mission planning and tracking demonstration for THETA. It is comprised of several managers listed below.

1.6.5.2.1 Bulletin Board Manager

The Bulletin Board Manager is part of the Mission Planning and Tracking demonstration application that was developed by Trusted Information Systems. This manager behaves like a mail program, but a user can only post a message to one of the predefined groups. See [38] for more information.

1.6.5.2.2 Downgrade Manager

The Downgrade Manager is part of the Mission Planning and Tracking demonstration application that was developed by Trusted Information Systems. This manager submits bulletin board

postings (that are managed by the Bulletin Board Manager) to the Regrade Manager to be reviewed. See [38] for more information.

1.6.5.2.3 Logistics Database Manager

The Logistics Database Manager is part of the Mission Planning and Tracking demonstration application that was developed by Trusted Information Systems. The database contains information about Air Force missions, flight plans, legs of flights, drop sites, etc. See [38] for more information.

1.6.5.2.4 Regrade Manager

The Regrade Manager is part of the Mission Planning and Tracking demonstration application that was developed by Trusted Information Systems. This manager cooperates with the Downgrade Manager and the Bulletin Board Manager in order to downgrade textual information safely. For a piece of information to be successfully downgraded, a privileged user must approve the submission. This privileged operation is regulated by the Regrade Manager. See [38] for more information.

1.6.5.3 Inventory Manager

The Inventory Manager maintains a database of information about stock supplies.

1.6.5.4 Mail Manager

The Mail Manager allows users to send textual information to other users. This manager operates at a single level only.

1.6.5.5 Set Manager

The Set Manager handles groups of object identifiers. These groups, known as *sets*, can contain object identifiers of any type. Standard set operations may be performed on set objects. For example, sets can be unioned and intersected, elements of a set can be added, deleted, etc. Also, each set may contain elements from lower security levels, thus making a "multilevel" object. Note that the object is not truly multilevel; the set object as a whole is marked at the highest security level, but each component of the object maintains its original security marking.

1.6.5.6 Thing Manager

The Thing Manager is a simple manager that implements and tests the operations on the basic object type "Object". The "Object" type is the superclass of all other types; that is, all types in

the THETA system inherit the attributes of the type "Object". The Thing Manager is used for verifying that the basic type and its operations are implemented correctly.

1.6.5.7 Tutorial Manager

The Tutorial Manager manages a test type created as part the *Manager Developer Tutorial for THETA* [32]. This very basic manager helps demonstrate some of the features of the THETA system and shows the developer some simple programming techniques of the system.

1.7 Summary

As explained in the previous portions of this document, THETA is a distributed, heterogeneous, secure operating system. As a distributed system, THETA has increased extensibility, availability, and resource sharing over centralized systems. Being heterogeneous, THETA provides greater availability to more hosts in a consistent manner. And, as a secure system, THETA protects data confidentiality and integrity.

An operating system provides abstractions by which applications may use, share, and control the resources of the underlying machine. THETA provides control of these COS resources in a distributed manner. A distributed operating system presents its users and applications with a set of uniform abstractions for the resources at multiple, independent processing locations. THETA is a secure distributed operating system that permits access to resources only if this access is consistent with a security policy.

THETA is intended particularly to support Air Force Command and Control (C^2) applications, though it is flexible enough to accommodate several other needs. C^2 applications provided several challenges. First, C^2 applications span many types of computer systems and require survivability, scalability, and interoperability. Second, they involve diverse aspects of the use of classified information including collection, selection, aggregation and analysis. Last, these applications involve monitoring and controlling physical devices that collect and use classified information.

Developing distributed services and applications is traditionally a very difficult exercise; however, THETA provides a suite of development tools that makes this task easier once they are trained in the art of programming in a secure OS environment. Development time is reduced since the programmer no longer needs to deal with the complexities of interprocess communication, interhost communication, data finding, access control, multitasking, and data storage.

THETA is an object-oriented environment by design. Because of its object-oriented nature, many of the hard concepts of distributed processing are abstracted away from the user, programmer, and administrator. Accesses to objects and services are consistent across all platforms on the THETA network.

1.7.1 Current Status

THETA is composed of several software components, a variety of hardware platforms, and operating systems that are native to the various machines.

The software components consist of the kernel, which provides the message passing facility between hosts and locally running processes; managers, which provide access to data objects in a regulated manner; and clients, which make access requests on objects by way of invoking operations on manager processes.

The hardware platforms presently maintained include the HP 700 series, Sun SPARCStations, and AT&T 386s. The processors used in these machine are PA-RISC chips, SPARC RISC chips, and 386 chips, respectively.

THETA runs on a variety of platforms, both trusted and untrusted. Though untrusted operating systems cannot support THETA functionality at the B3 level, they can still be included in a network of THETA machines as long as they are run at a single level. THETA is supported on the following operating systems:

- HP-UX BLS 8.09+ - This system is designed to meet the TCSEC level B1. THETA 1.5 (excluding TNET) and 1.7b (excluding TNET) run on this system.
- HP-UX 8.09 - This system is an untrusted operating system and does not meet standards specified in TCSEC. THETA 1.5 (excluding TNET) and 1.7b (excluding TNET) run on this system.
- Sun CMW 1.0 - The Sun Compartmented Mode Workstation is designed to meet the Compartmented Mode Workstation requirements, which are similar to requirements for TCSEC B1 systems; however, Compartmented Mode Workstations have additional criteria concerning windowing environments. See [43] in "References" on page 91. THETA 1.5 (including TNET) and 1.7b (excluding TNET) run on this system.
- SunOS 4.1.X - This system is an untrusted operating system and does not meet standards specified in TCSEC. THETA 1.5 (including TNET), 1.7b (excluding TNET), and 2.3 (excluding TNET) run on this system..
- AT&T System V MLS - This system is designed to meet TCSEC B1 criteria. THETA 1.3a (excluding TNET) runs on this system.
- Trusted Solaris 1.2 - This system is designed to meet TCSEC B1 criteria. THETA 2.3 (excluding TNET) runs on this system.

1.7.2 Future Plans

There are three major goals to be met in the long-term. We wish to create a successful demonstration and research testbed in the government community, to refine THETA to a production-quality development environment, and to conform to emerging commercial standards on object-oriented technology.

An immediate goal has been inspired by the Joint Directors of Laboratories (JDL) Security Evaluation Program (SEP). THETA is becoming a rich demonstration and research testbed. The framework is being put in place for each military service to produce their own THETA services and applications with the goal of interoperating securely over a wide-area network with the other military services. This experiment will demonstrate the feasibility of realistic THETA applications and will test the usability of the THETA development environment.

Development of THETA began as an experiment in secure distributed networking. We feel the research has been successful; however, the resulting system is not yet polished enough to consider it production-quality. To provide a foundation for the previously mentioned goal, we plan to enhance the administrative tools to support multicluster operation of separate administrative domains. We also plan to increase large-scale wide-area network support in the areas of secure database interfaces, auditing, replication, invocation tracking, multilevel atomic transactions, and network packet encryption.

Last, we are working towards compatibility with commercial distributed computing standards. As a member of the Object Management Group, we are actively participating in defining security specifications for the Common Object Request Broker Architecture (CORBA) [17]. Because THETA's architecture and philosophy are very similar to CORBA's, we believe that a CORBA-compliant version of THETA is the logical next step. We foresee THETA-CORBA interoperability to be an attainable goal, and with that, we achieve accessibility for THETA within the commercial marketplace.

1.7.3 Published Papers

For more information on various aspects of THETA, this section lists the papers that have been published as a result of the THETA research project.

- McCullough, D. "A Hookup Theorem for Multilevel Security", *IEEE Transactions on Software Engineering*, 16(6):563-568, June 1990.
- McCullough, D. "Foundations of Ulysses: The Theory of Security", Technical Report RADC-TR-87-222, Rome Air Force Development Center, May 1988.
- McEnerney, J., Weber, D., Browne, R., and Varadarajan, R. "Automated Extensibility in THETA" *Proceedings of the 13th National Computer Security Conference*, October 1990.
- Pascale, R., and McEnerney, J. "Using THETA to Implement Access Controls for Separation of Duties", *Proceedings of the 17th National Computer Security Conference*, 1994.
- Proctor, N., and Wong, R. "The Security Policy of the SDOS Prototype", *Proceedings of the 5th Annual Computer Security Applications Conference*, December 1989.
- Seager, M., Guaspari, D., Stillerman, M., and Marceau, C. "Formal Methods in the THETA Kernel", *Proceedings of the Symposium on Research in Security and Privacy*, May 1995.
- Varadarajan, R., et al. "SDOS–An Overview", *1989 Mission Critical Operating Systems Workshop*, September 1989.
- Weber, D. G., and Lubarsky, R. S. "The SDOS Project - Verifying Hook-up Security", *Proceedings of the 12th National Computer Security Conference*, October 1989.

- Wong, R., et al. "The SDOS System: A Secure Distributed Operating System Prototype", *Proceedings of the 12th National Computer Security Conference*, October 1989.
- "Application of Formal Methods", edited by Hinchey and Bowen, Prentice Hall, 1995, pp. 285-306.

2 THETA and CORBA Comparison

The Common Object Request Broker Architecture (CORBA) specification is a rapidly emerging standard that appears particularly relevant to THETA. In this chapter we provide a comparison of THETA and CORBA, and judge the potential for a CORBA-compliant THETA. We first provide some background on the Object Management Group (OMG) and give an overview of the Object Management Architecture (OMA). We then discuss THETA as an example of an OMA and describe how THETA components map to the OMA. Next we compare the evolving security requirements of CORBA with the THETA security architecture. We then describe the steps required to develop a CORBA-compliant THETA, and conclude with a discussion of the potential for such a system in non-DoD applications.

2.1 CORBA Overview

2.1.1 Object Management Group

The OMG, consisting of over 500 software vendors, software developers, and end users, is the world's largest software development consortium. All of the large computer vendors are represented, including IBM and Microsoft. The mission of OMG is to promote the development of object technology for distributed computing systems. The goal of OMG is to provide a common architecture framework for object-oriented applications based on widely available interface specifications.

In 1991, the OMG released the first version of the CORBA specification, which defines the architecture of the Object Request Broker (ORB). The ORB provides the interoperability mechanisms that allow objects and applications to communicate in a heterogeneous distributed environment. The ORB is part of the overall OMA, as described in Section 2.1.2. The ORB and OMA specifications continue to be refined and updated by adding new functionality and features, including security.

Responding to a growing need for security in distributed systems, the OMG Object Services Task Force Security Working Group released a Request For Proposal for CORBA Security Services (known as OSTF RFP3) [15]. OSTF RFP3 has a very flexible view of security. The OMG would like CORBA-compliant systems to be able to support a wide variety of security policies, from high-assurance DoD access control to relatively weak mechanisms required in

many commercial applications. The challenge of considering THETA in this context is to investigate whether it is feasible for THETA to maintain its Trusted Computing Base (TCB) architecture for providing assurance, yet still be interoperable with CORBA-compliant products.

2.1.2 Object Management Architecture

The OMA defines the overall OMG view of an object-based distributed environment [14]. As shown in Figure 2-1, there are four main pieces to the OMA: the ORB, Application Objects, Object Services, and Common Facilities. The ORB provides the uniform interface that allows objects to interact regardless of the programming language, operating system, hardware, or network. End-user application client and server programs are defined as Application Objects within the OMA. The Object Services are the objects that perform low-level fundamental system operations, such as namespace and persistence services. Object Services are typically supplied by the ORB vendor. Finally, Common Facilities are application-level functions that are common across many users, such as printing, databases, and compound documents.

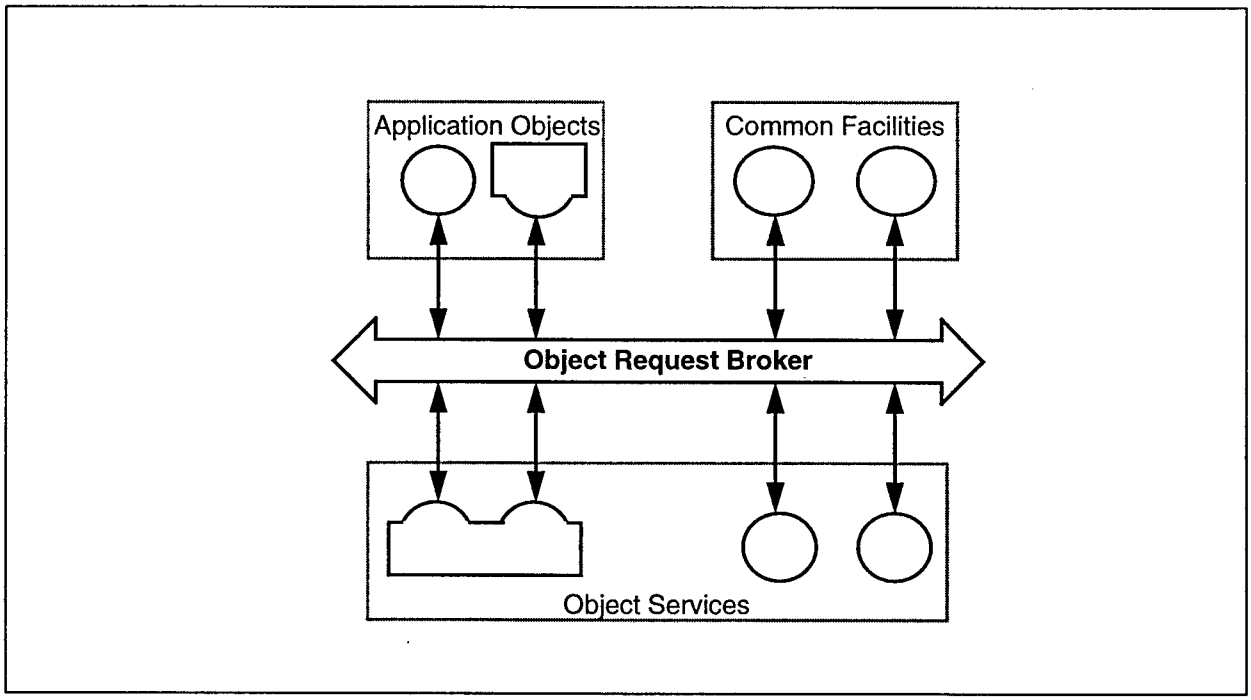

Figure 2-1: Object Management Architecture

All communication among Application Objects, Object Services, and Common Facilities is through the ORB. The structure of the ORB is shown in Figure 2-2. The ORB is responsible for handling all requests that are sent by a client to an object implementation, as well as any subsequent replies back to the client [17]. The ORB must find the appropriate object imple-

mentation for the request, get the object implementation ready to handle the request, and then ensure that the request data is transmitted to the object implementation.

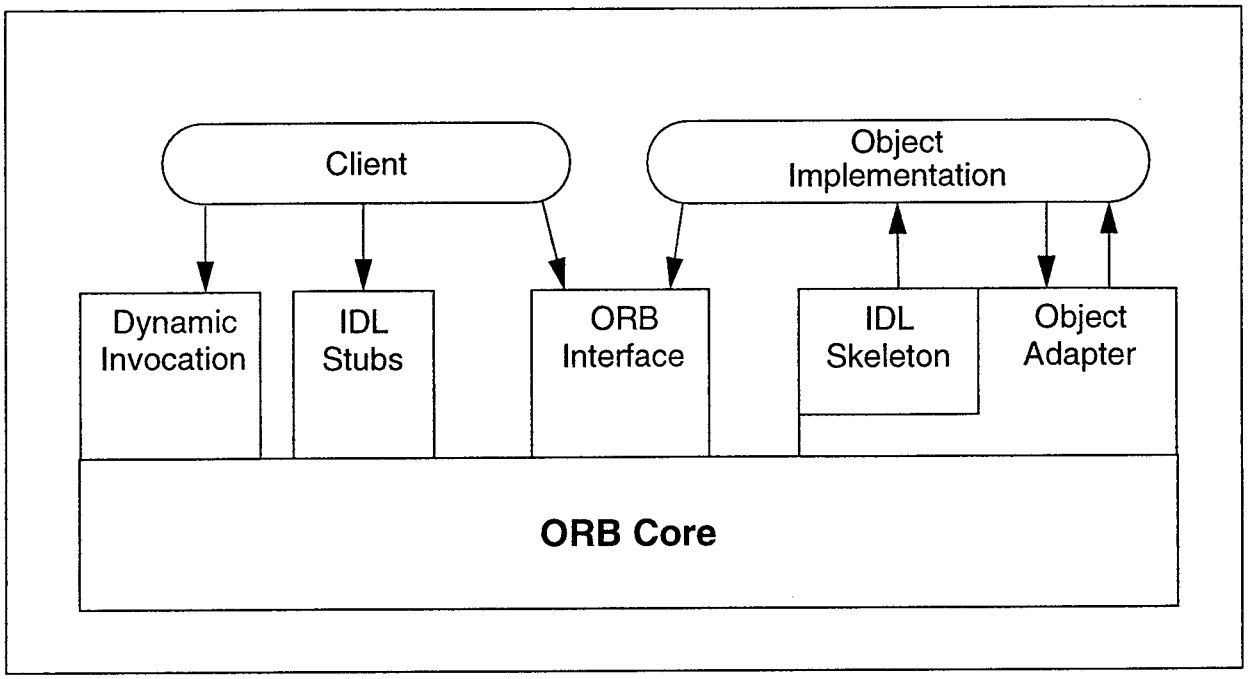

Figure 2-2: Structure of ORB Interfaces

For a client to make a request to an object implementation, the client can use either an Interface Definition Language (IDL) stub or the Dynamic Invocation interface. An IDL stub is a statically defined interface that is compiled into the client code. Clients can also access object interfaces at runtime through the Dynamic Invocation interface to construct requests on the fly. These object interfaces are stored in the Interface Repository. The object implementation receives a request from the client in the form of an up-call through the IDL skeleton.

IDL defines the interface to objects by describing object operations as well as the parameters to those operations. Tools supplied by the vendor along with the ORB translate IDL specifications into the IDL stubs for use by clients as well as the IDL skeletons for use by object implementations. In this manner, clients and object implementations can communicate through the IDL-defined interface even when running on different platforms.

The object adapter provides the interface for ORB services to the object implementation. ORB services provided by the object adapter typically include: handling of object references (unambiguous object identifiers), method invocation, security, and registration, among others. There may be many varieties of object adaptors to support various specialized object implementations, such as OO database adapters. The Basic Object Adapter (BOA) is available on every ORB implementation and provides a general set of functions that are useful for many object implementations.

The ORB interface is the same for all ORBs and provides a direct interface to a few operations that are common across all objects. These operations may be used by both clients and implementation objects.

The ORB core is the inner ORB component that moves the request from the client to the object adapter that is appropriate for the target object implementation.

2.2 THETA as an Object Management Architecture

With the exception of THETA security mechanisms, the structure of THETA and CORBA are very similar. In this section, we make a general comparison of THETA and CORBA functionality without considering the security of each architecture. We describe the mapping of THETA components to the OMA. The general functional comparison in this section provides the background to address THETA and CORBA security issues in Section 2.3.

The mapping of THETA components to the OMA is straightforward. Since both are distributed OO architectures that are designed to work in heterogeneous environments, they necessarily contain many of the features. Furthermore, THETA is derived from BBN's Cronus, which in turn was driven by many of the same ideas that originally motivated the authors of CORBA.

The basic object models of CORBA and THETA are identical. The concepts of clients, objects, requests, types, interfaces, and operations are equivalent. CORBA object references correspond to THETA unique identifiers. THETA objects are accessed through managers. This implementation style corresponds to the CORBA BOA implementation style called the persistent server activation policy. THETA also supports object replication, which is planned for a future version of CORBA.

As described previously, the principle components of the OMA are the ORB, Application Objects, Object Services, and Common Facilities. In THETA, ORB functions are provided by a combination of the THETA Kernel, portions of the managers, and the Program Support Library (PSL). OMA Application Objects correspond to THETA application clients and managers (e.g., the Mission Planning and Tracking application implemented by Trusted Information Systems). OMA Object Services correspond to several of the THETA system managers, including the Authentication Manager, the Configuration Manager, the Directory Manager, the Primal Process Manager, and the THETA File Manager. Both the OMA and THETA provide support for object persistence and name services. OMA Common Facilities correspond to the application-level THETA system managers, such as the Audit Manager and Automatic Code Generator (ACG) Manager.

The CORBA IDL is roughly equivalent to the THETA type and manager specifications. Although the language syntax is different, both define object types, operations, and parameters, and provide class inheritance. CORBA IDL supports multiple inheritance, while THETA

type specifications provide single inheritance. IDL supports a number of mappings in implementation languages, while THETA specifications only map to C. Like CORBA, THETA generates (using the ACG manager) client stub and object implementation skeleton code. The ACG manager also provides a facility similar to the CORBA Interface Repository by making object interfaces available at runtime. The mapping of the THETA architecture onto the ORB interfaces is shown below in Figure 2-3.

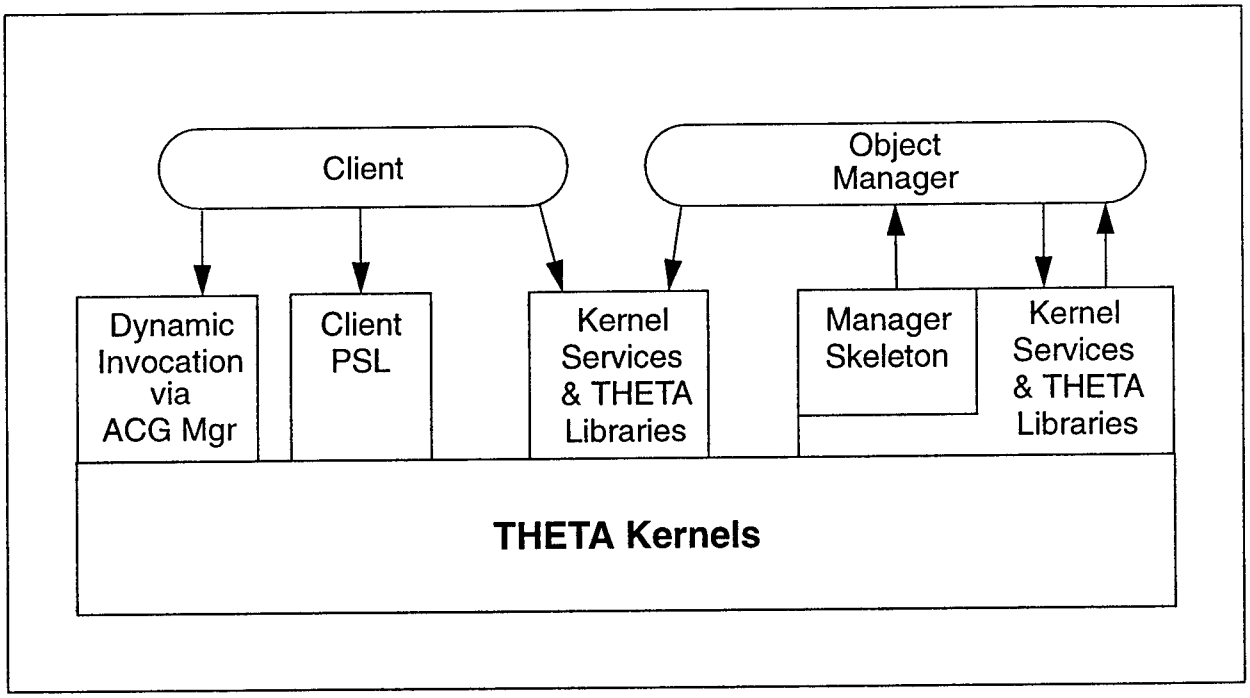

Figure 2-3: Mapping of THETA Architecture onto ORB Interfaces

CORBA object adapter and ORB interface functions are provided in THETA by a combination of THETA Kernel services and THETA library code. Within this code, services are provided for handling unique object identifiers, method invocation, security checks (discussed further in the Section 2.3), and registration. Both CORBA and THETA support synchronous remote procedure call (RPC) style invocations, although THETA also supports asynchronous invocations (planned for a future version of CORBA).

Finally, the ORB core corresponds to the set of THETA Kernels running on various hosts, which handle dynamic object location and transmission of the request from client to manager.

In summary, the architectures of CORBA and THETA are already very similar. The primary differences are due primarily to choices in the style of interfaces rather than distinctions in the conceptual object model.

2.3 Security Requirements of CORBA vs. THETA

The greatest difference between CORBA as it now exists and THETA is in the area of security. The current definition of CORBA is essentially silent on the topic of security. Although some ORB vendors provide basic security mechanisms, there has been no consistent approach to security within the OMA. The OMG recognized this serious deficiency, and as a result released OSTF RFP3 [15], which asks for vendors to propose general solutions for security within the OMA. A set of three submissions by major vendors and their partners were merged into one security standard known as the CORBA Security Standard. The standard was voted on and accepted in January, 1996. ORA's role in the standard was to contribute to group discussions and to write two appendices--the conformance evaluation and vulnerabilities and threats.

Although the security requirements for CORBA have not been conclusively defined, it is definitely not too early to plan how THETA security relates to CORBA. The existing submissions for RFP3 are sufficiently similar that there is a good indication of what security means for CORBA. Furthermore, THETA serves as an excellent yardstick for judging the adequacy of CORBA security requirements. If a system similar to THETA cannot be implemented within the CORBA guidelines, then we believe that CORBA will not be capable of supporting high-assurance MLS policies. As a co-author of a submission for OSTF RFP3, ORA has already used THETA as a guide to justify requirements for CORBA security features.

THETA can also drive CORBA security by continuing to stay at least one step ahead of the security defined in existing OMG standards. By serving as a demonstration vehicle for advanced security technology, THETA can help OMG avoid poor security design choices and thus facilitate emergence of high quality commercial OO distributed security products.

In this section, we first give an overview of the direction of OSTF RFP3 and then describe how THETA security relates to this evolving standard. We do not compare and contrast the details of each of the OSTF RFP3 submission because these details are currently subject to change. Instead, we give general impressions of the security requirements of RFP3, particularly as they relate to assurance.

2.3.1 Object Services Task Force RFP3

The philosophy of the final OSTF RFP3 submission is likely to emphasize flexibility. Consistent with the approach of other OMG documents, RFP3 will be inclusive of many variations of security architectures rather than exclusive. This style of specification is driven by market demand—vendors don't want to be forced into highly constraining security architectures because many end-users think that security gets in the way of accomplishing their work. At the same time, vendors perceive that in many vertical markets (e.g., finance, healthcare) there is a growing demand for better distributed security mechanisms, and they know that CORBA must address these needs in order to be competitive.

As a result, RFP3 submissions are described in terms of a security framework. The framework is sufficiently general to allow confidentiality, integrity, accountability, and availability policies for a broad array of vertical markets. The framework provides this flexibility by allowing customization of the access checking code within the ORB, Object Services, Client, and Object Implementation. In this manner, the submission attempts to address both DoD MLS security requirements as well as much less stringent requirements in areas such as business automation.

An illustrative candidate security reference model is shown below in Figure 2-4. The model depicted provides a simple framework for many different access control security policies. This framework consists of two layers: an *application access policy* and an *object invocation access policy.*

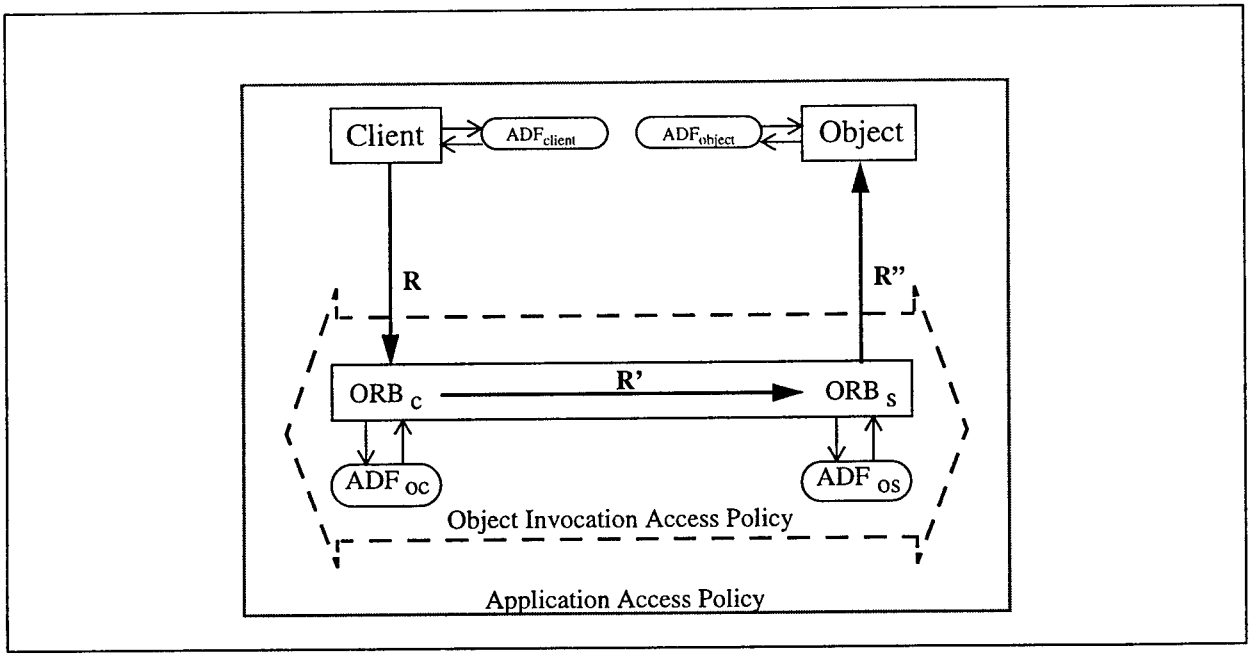

Figure 2-4: CORBA Access Control Model

The application access policy governs the performance of object operations on behalf of clients. The application access policy is enforced within the client and/or the object implementation. If the client or object enforce such a policy, then that portion of the system is inherently trusted to correctly implement and enforce the policy. Note that a valid instantiation of this framework is for both client and object to be security unaware; in this case all mediation would be performed by the ORB.

The object invocation access policy governs the delivery of messages between authenticated clients and objects. This policy is enforced within the ORB although mediation decisions and support may be carried out by other portions of the OMA, in particular the Object Services. All instantiations of the security reference model place at least some trust in the ORB to

enforce the object invocation access policy. Even in architectures where the access control mediation occurs solely within the client and object, the ORB is still required to validate the request parameters and ensure message delivery. Mediation of object replies is similar to requests, so for simplicity we do not explicitly address replies as a special case.

The access control model shows the client invoking an operation as specified in the request **R**. The client first tests the request against the application access policy (specified by the client's Access Decision Function, i.e., $\mathbf{ADF_{client}}$) before passing the request to the ORB. $\mathbf{ORB_c}$ then tests the request against the client side object invocation policy ($\mathbf{ADF_{oc}}$), and if the test succeeds, transforms **R** to **R'** and passes the request **R'** to $\mathbf{ORB_s}$. $\mathbf{ORB_s}$ then tests the request against the server side object invocation policy ($\mathbf{ADF_{os}}$), and if the test succeeds, transforms **R'** to **R"** and passes the request **R"** to the object. The object finally tests the request **R"** against the application access policy ($\mathbf{ADF_{object}}$) before executing the method specified in the request. By instantiating ADF_{client}, ADF_{object}, ADF_{oc}, and ADF_{os} differently, this framework could support many different policies. For example, ADF_{oc}, and ADF_{os} could be defined to enforce an MLS property in some systems, or an ACL access check in others.

The flexibility of a security framework comes at the cost of complexity. Although a flexible architecture potentially allows developers to customize security for a wide variety of markets, the complexity of this approach could increase ORB vendor cost and risk. For this reason, most initial instantiations of CORBA security are likely to be systems with modest built-in authorization and authentication controls.

In addition, a framework of this complexity could jeopardize applications requiring assurance such as MLS. Typical security evaluation tasks, such as identifying the TCB boundary and the security reliance on the underlying OS and network, could become very difficult when security mechanisms are potentially distributed throughout the CORBA security architecture.

2.3.2 Relationship of THETA to CORBA OSTF RFP3

The CORBA security framework appears to be sufficiently general to support the THETA security policy. The CORBA object access policy ($\mathbf{ADF_{oc}}$ and $\mathbf{ADF_{os}}$) would support the THETA MAC policy enforced by the THETA Kernel, while the CORBA application access policy ($\mathbf{ADF_{client}}$ and $\mathbf{ADF_{object}}$) would support the THETA DAC policy enforced by THETA managers.

Our experience with the THETA architecture and its assurance argument is directly relevant to the CORBA framework. In THETA, access checking code is carefully structured to minimize trusted code while still allowing extensible application security policies. To have a valid architecture in CORBA, the same argument would need to be made. Thus, it must be possible to structure the ORB so that it contains a minimum of trusted code. It must also be feasible to use the protection mechanisms of the underlying operating system and hardware to demonstrate

that this distributed TCB has all of the usual properties of a reference monitor: it must be tamperproof, always invoked, and sufficiently simple to be subject to thorough test and analysis.

Although the OSTF RFP3 submission definitely does not require that this structure exist in all CORBA security architectures, so far we have seen nothing in the RFP3 submissions that prohibit this style of structure and analysis.

2.4 Steps to Reach a CORBA-Compliant THETA

CORBA compliance is the next logical step for THETA development. Proceeding in this direction meets several objectives. As described earlier, THETA can encourage high quality commercial OO distributed security products by serving as a demonstration vehicle for advanced security technology. Given the strong commercial market motivation of OMG, it is unlikely that any ORB vendors will be willing to risk the investment to build an MLS distributed ORB in the near term. THETA development may be the most effective way to demonstrate MLS distributed OS technology. CORBA compliance is major step toward interoperation of THETA with commercial distributed systems, thus allowing experimentation with interoperability between trusted and untrusted domains. Finally CORBA compliance will serve to mature the THETA environment. OO technology has evolved significantly since it was used to produce Cronus and THETA. By building a CORBA-compliant THETA, we will replace the OO model in THETA with the much more clean and elegant model contained within CORBA.

In the remainder of this section, we outline the steps suggested to develop a CORBA-compliant THETA.

2.4.1 Object Interface

The first phase of development would be to produce a CORBA-compliant interface as specified in [17] and OSTF RFP3. This effort would require interface modifications to the Kernel, THETA libraries, and the ACG manager to match the CORBA interface specifications for the Dynamic Invocation interface, the IDL stubs, ORB interface, IDL skeleton, and Basic Object Adapter. As part of the ACG modifications, it may be feasible to use Sun's public domain front-end to parse CORBA IDL.

The largest part of this effort would be to update the existing THETA system and application managers and clients to use this new interface. This effort would not be difficult, but it would be very time-consuming due to the very large amount of existing manager code. Updates should be prioritized to ensure that the most critical system managers (i.e., authentication and primal process) are running first to allow early experimentation.

The security policy enforced in this phase would be essentially identical to the current global THETA DAC and MAC mechanisms.

2.4.2 Secure Interoperability

The second phase of this effort would be to address secure interoperability between a CORBA-compliant THETA and a commercial ORB product. Because secure interoperability standards are unlikely to be resolved within the RFP3 submission, this effort would serve as a feasibility demonstration for OMG for how secure interoperability could be supported. The interoperability protocol could be based either on extensions to the Internet Inter-ORB Protocol (IIOP) or the DCE Inter-ORB protocol [16]. The access control policy supported across the ORBs would handle a fixed discretionary access control policy and mutual authentication.

2.4.3 Extensible Policies

The third development phase would be to support extensible authentication and authorization policies, as well as multiple policy, trust, and technology domains. These capabilities will be documented in the RFP3 submission, but early secure ORBs are unlikely to fully exercise this capability. Extensible access policies would allow developers to tailor ORB security to specific applications. Security domains allow secure interoperation across separate enterprise and administrative organizations. THETA development in this area would be to demonstrate to vendors that such higher-risk implementations were feasible, thus encouraging faster development of commercial ORB products designed for multipolicies.

2.5 Potential for THETA Use in Non-DoD Applications

THETA security policy is specifically designed to support high-assurance military applications. Commercial users are not likely to be willing to run on MLS platforms to obtain the additional protection assurance that is available in that environment.

Although non-military use of THETA as it exists is doubtful, there are two principle ways that THETA technology can support non-DoD applications. First, THETA technology can be used to drive CORBA security, as discussed earlier. Because of the widespread interest in using CORBA for commercial applications within the OMG user community, getting THETA security mechanisms adapted within OMG is one of the most effective ways to result in non-DoD use of THETA. Second, developing a CORBA-compliant THETA with an extensible policy opens many avenues for commercial uses. By providing customized security services beyond MLS (e.g., integrity, role-based access control) as well as interoperability to other ORB platforms, THETA could serve as an "intelligent" firewall for restricting access between two ORB domains.

3 THETA and DCE

3.1 Introduction

THETA and the Open Software Foundation (OSF) Distributed Computing Environment (DCE) [21] are both systems of distributed client/server application programs which can run on a variety of underlying operating system and hardware. There are a number of overall similarities between the two systems. Both consist of:

- An Interface Description Language (IDL) and IDL tools which generate client and server stub code;
- Supporting servers which provide user authentication and object naming and location;
- Supporting software which provides a Remote Procedure Call (RPC) infrastructure for client/server communication, and which supports access control by encoding client user identities;
- Supporting server software which performs access control on objects;
- Supporting server software which enables multi-threading;
- A file server.

While THETA's architecture does not specifically call out the above as components, DCE's architecture is more oriented toward these functional areas. DCE consists of the following capabilities:

Remote Procedure Call (RPC): a runtime system consisting of library software that uses network protocols and, optionally, an RPC server and security and directory services. All DCE components interact with one another and with clients via RPC. DCE RPC is compared with THETA in Section 3.5, along with the security and directory services.

Security: a service consisting of two parts: the Security Server which allows clients and servers to authenticate and obtain identifying credentials; and part of the RPC mechanism whereby these credentials can then be passed in RPCs so that the recipient can identify the sender. In the case of servers which perform access control, the client credentials are compared with the Access Control List (ACL) appropriate to the client's request. Security service is described further in Sections 3.4 and 3.5.

Directory Service: performs object name resolution and location for callers of RPCs. Though not strictly required for RPC, Directory service is closely related to RPC, as described in Section 3.2.

Distributed File System (DFS): The DFS is a DCE application built upon the infrastructure of the other DCE components. Unlike the other components which are interdependent to some degree, DFS is not used by any other component and as such is in principle simply one of many possible distributed client/server applications built with DCE. The THETA file manager has exactly the same role in THETA as DFS in DCE. It is not an essential THETA server, in that THETA developers may, but need not, use the file manager for storage service that is independent of COS-specific file service APIs. In practice, DFS has not been widely used, partly because it is only just becoming fully available, and partly because it is only just gaining commercial-quality robustness and functionality. DFS has a richer set of functionality than the THETA file manager, due to its evolution from the Andrew File System (AFS) and due to its commercial orientation. Such functionality includes: a separate local file system for efficient local access and quick recovery; sophisticated client-side software for caching and cache consistency; file grouping for cloning, relocation, quota restrictions, backup/restore, and replication; and Unix file system interoperability. Although DFS's replication mechanism is not as flexible as THETA's (DFS uses master/slave arrangement with a specific single-image consistency policy) DFS replication is tuned to its other areas of functionality (e.g., fault recovery, relocation). This document says no more about DFS.

Time: a service provided by the DCE time server, and client software for accessing the server. The purpose is to provide a common time frame for servers on a common LAN. DCE directory and security servers use time for coordinating and sequencing shared computation. Time service is also available to new applications with common timing needs. There is no comparable service in THETA, and THETA components have no requirements for common time. This document says no more about the DCE Time service.

Threads: a library which implements multiple threads of execution within one address space. The DCE Thread package is largely based on POSIX Pthreads, but is intended to provide a common API across multiple heterogeneous O/Ss, some of which may not support Pthreads. The DCE Thread package is similar in function to THETA's manager tasking package, except that the latter is more tightly coupled to the THETA manager architecture, and is used only in managers. DCE Threads are not an integral part of DCE servers, but may be used by both clients and servers.

3.2 Object Naming, Location, and Invocation

The different object models of THETA and DCE are fundamental to other differences.

In THETA, all a manager's resources are divided among discrete objects which are separately identified (using a globally unique identifier, or UID) and access controlled. As a separate matter, objects may be given names; the directory manager keeps a mapping from items in a usual hierarchical name space to the UID of each item. Accessing an object by name is actually two separate steps: first contacting the directory manager to obtain the UID of a name; and second, using the UID to request an operation on the object. If a client already has the UID, the first name resolution step is not necessary. The directory manager, of course, has no name, and must be contacted via its UID. Such UIDs, which denote an entire type (as services are called in THETA) are called generic UIDs.

THETA object location is entirely based on UIDs and hence is divorced from naming issues. Object location is performed by the THETA kernel as part of its central function of switching all messages from the sender to the recipient. Each THETA message has a UID as its target, and the THETA kernel is responsible for locating the target object denoted by the UID. Some location operations involve searching for the target object on other hosts, by sending location messages to location components on other hosts. The resulting remote location data are kept in a cache for later use. Therefore, the only location operations that require remote searching are those in which the target object is not in the location cache.

However, the details of this location activity are invisible to THETA message senders. In every case, the sender gives the message to the THETA kernel, which locates the target object and forwards the message to the recipient associated with the target, e.g. the object manager that manages the object denoted by the UID.

In DCE, naming and location are more tightly entwined. Each DCE service has both a hierarchical name and a globally unique identifier, or UUID. Thus a DCE service UUID is analogous to a THETA generic UID. Below the service API, client stub code contacts the Cell Directory Server (CDS) to determine how to contact a server that provides the desired service. For the server that implements the RPC called by the client, the stub code specifies the server by UUID. However, clients may also specify servers by name– for example, when using security services.

Whether given a name or UUID, the CDS looks up the host-ID of a server running the requested service. This host-ID is obtained not by searching for a host running the service, as the THETA locator does. Rather, the CDS will have already obtained the host-ID from the server itself: it is the responsibility of each DCE server to register itself with the CDS. The CDS returns host-IDs to callers, which use the host-ID to contact a server directly. However, the caller needs more information that the host-ID; it also needs a communication endpoint on that host. Therefore the RPC caller consults the endpoint map on the host specified by the CDS.

The endpoint map is maintained by the RPC server. The CDS and RPC server are separate because endpoint mappings are specific to each individual host, and as a result, there must be a RPC server on each host. The CDS's mapping between names and hosts is, in contrast, glo-

bally meaningful data, and as a result may kept by a single CDS rather than requiring a CDS in each host. (Although CDSs may be replicated for high availability, the cost of a replica on each host would outweigh the benefit in a but the smallest cells).

The nature of an endpoint is dependent on the network transport mechanisms supported by the server. However, in most or all DCE implementations an endpoint is a TCP or UDP port. Having obtained the endpoint, the caller can then connect to the server and send the RPC message directly to it. In addition, the binding is retained so that subsequent messages can be sent directly to the server without the intervention of the CDS and the RPC server.

Note that there must be a method for clients to bind to the RPC server, which in turns tells the client how to bind to other servers. This is accomplished by the simple means of the RPC server using a single well-known port; clients can bind to that port without any further information.

In summary, there are several differences between DCE and THETA in the area of naming and location:

- In THETA names and UIDs are separately used, with only UIDs being relevant to location of an object. In DCE both names and UIDs are identifiers used by the CDS.
- In THETA, there is a locator component on every host. In DCE the primary location component, the CDS, runs only on one or a few hosts.
- In THETA, once an object has been located, the THETA kernel sends the message to its destination, using THETA internal communication mechanisms built on common network protocols. In DCE, the location is returned to the caller, which uses it to contact the server directly using common network protocols.
- In THETA, each operation is targeted at a specifically identified object, which the THETA kernel must locate in order to forward the operation message. Any optimizations of location processing (e.g. caching of location data) are the responsibility of the THETA kernel. In DCE, services are named and located, but subsequent RPCs can be targeted to a specific previously located server to which the client has a direct communication channel. Optimizations of RPC traffic (e.g. caching of communication channels for later RPCs) are the responsibility of the DCE runtime software that is part of every DCE client or server.

With respect to the last point, it should be pointed out that DCE servers may function as true object managers in the manner of THETA, with individually named and uniquely identified objects and with RPCs that implement operations on an object. Although this approach is supported by DCE's notion of object UUIDs, this approach is not part of DCE mechanisms. Further DCE support for an object model (similar to that of THETA) is provided by CORBA systems built on DCE.

Despite these differences in mechanism, both DCE and THETA provide the same basic functionality of transparent distributed access to remote objects. In THETA, the transparency is implemented by one component, the THETA kernel, which performs all location functionality, hides from both the client and server their relative locations, and thereby hides the dynamicism of the distributed environment in which objects and/or services may move. The same in true in DCE, but transparency is implemented by the combination of the RPC server, the CDS, and the DCE runtime software's use of them. That is, a DCE service's client stubs (the equivalent of THETA PSL calls) use a library of DCE runtime software both to call on the RPC server and CDS to locate a server, and to send the RPC to the server. In THETA, the analogous functionality is concentrated in the THETA kernel, which handles all location and message transmission on behalf of THETA processes.

Appendix A provides a step-by-step summary of DCE location and invocation mechanisms, and the relation to the use of credentials for secure RPCs.

3.3 Mandatory Policies and Communication

Another key point about DCE location is that it is *not required* for client/server communication via RPC. A client and a server may rendezvous by purely conventional means, e.g. by the client's including the knowledge of the server, and the server being available for binding at a well-known port known to the client. CDS and RPC services merely provide a flexible common mechanism for clients to bind to servers, but any client/server application could implement its own mechanism.

Therefore, the most significant difference between DCE and THETA is that DCE has no mechanism for mediating the communication between clients and servers, as there is in THETA. Any attempt to add mediation functionality to the CDS and the RPC server would be fruitless because of their optional role in client/server communication. In THETA, by contrast, the THETA kernel switches all messages[1] and implements a mandatory policy of information flow.

Note that such an architecture is not the only means of implementing a mandatory policy. Another approach is for client/server communication to be by means of protected capabilities which are doled out by the TCB in accordance with a policy, but which can then be used for direct communication. Such an approach is equally inapplicable to current DCE implementations, because endpoint bindings are simply numbers (TCP or UDP ports) rather than protected capabilities.

---

[1] An exception to message switching is direct connections between clients and servers, but the setup of these is also mediated by the THETA kernel.

Because of the lack of multi-level and/or capability-based network protocols to underlie client/server communication, there does not appear to be any feasible means of performing THETA-style message policies in a DCE system.

3.4 Access Control

Besides implementing a mandatory sensitivity policy on client/server communication, THETA also provides similar mandatory and discretionary policies on objects, using label-based and ACL-based access control mechanisms. As described above, DCE services need not have a similar object model, so access controls of DCE servers need not be oriented towards objects. DCE does provide a server framework which includes an ACL manager component which can implement the responsibility for access controls based on ACLs. However, there is considerable latitude concerning what DCE ACLs specify the access to. Each THETA ACL, in contrast, always describes access to a THETA object.

DCE does not have a general-purpose ACL manager. Different servers implementing different services may need different types of ACLs, e.g. with different modes. If an application developer's needs are not met by an existing ACL manager, then the application may need to develop a new one. THETA, by contrast, has a single, general, extensible ACL component of the general THETA manager skeleton. This ACL mechanism can be extended or customized by statements in the THETA IDL e.g. those defining new access modes.

Additionally, DCE does not have a label-checking access control module analogous to the ACL manager. However, the DCE ACL mechanism is general enough that a specific server could implement an ACL manager that performs MAC checks as well. Note that such an approach, applied to a DCE server that imposes access controls on a per-object basis, could yield DCE servers that function in a manner similar to THETA MLS managers with a range of system-low to system-high. However, because of the lack of message mediation, there is no potential DCE equivalent of multiple single level (MSL) servers, or MLS servers with a limited range. (For MSL managers, THETA message mediation functions to constrain each individual manager's operation to a single level where mandatory mediation by the manager is not needed. The same is true for limiting partial-range MLS managers' operation to the range within which it does perform mandatory mediation.)

3.5 Secure Remote Procedure Call

DCE security can be used with DCE RPC to implement a secure RPC mechanism that is similar to messaging in THETA. Note, however, that security is an option; clients can choose whether or not to authenticate their messages, and servers can choose whether to require that messages be authenticated.

As with THETA messages, DCE secure RPC messages contain information identifying the sender of the message. In THETA messages, however, the sender identity is both unprotected and implicitly vouched for by the THETA component sending the message. There are different mechanisms for assuring the correctness of message sender identity, in local and network messages. For THETA local messages, there is the assurance that the message was received by the client or server from the THETA system itself, which is trusted with respect to correct identification and authentication. For THETA network messages, assurance of correct sender identity comes from application-level cryptographic protection. This protection ensures the authenticity of the remote THETA system as the conveyor of the message, and ensures the integrity of the message and the message sender indentity enclosed in it. Message privacy is also an option, on a host-to-host basis.

Thus, correct message authentication and operation authorization depends on (a) secure operation of THETA on each host, to avoid local spoofing and snooping, and (b) secure key management and protection of each host's cryptographic key used to ensure message security.

In other words, THETA uses authenticators (data that describes an identity) that consist of some plaintext in THETA messages; THETA uses local host trust and host-to-host cryptography to protect interhost message streams as a whole. Specifically, the plaintext is the UID of the principal associated with a client request or server reply.

In contrast, DCE uses cryptographic authenticators in the manner of the Kerberos system. Individual messages contain authenticators which can be subjected to cryptographic analysis to determine veracity. In addition, the cryptographic techniques used to protect the message authenticator may also be used to protect the privacy and/or integrity of the message as a whole.

In other words, DCE uses message-level cryptography to protect individual authenticators and, optionally, individual messages.

Note, however, that the security of DCE messages rests on storage of individual client session keys in the local host operating system. The same is true of the THETA key which is used to encrypt traffic between THETA kernels.[2] Fundamentally, both system's security rests on the ability of the host O/S to protect key data. The main difference is that with DCE the domain of usage of each keys is much smaller, and the results of compromise more limited. That is, each key stored on a client host is a session key for one client session. For THETA, each key is used for traffic between all THETA kernels. In addition, the key management in THETA is *ad hoc*.

Therefore, THETA's cryptosystem does not scale as well as DCE's Kerberos approach (although Kerberos has scaling problems as well). The two cryptosystems have comparable

---

[2] Actually, there is more than one key. For each security level in the system, there is a key that all hosts share to protect traffic at that level.

risks with respect to reliance on a base O/S for key protection, but in THETA the risks become more concentrated as systems scale up in size.

Appendix B provides a high-level summary of the cryptographic credentials that DCE uses as authenticators. Appendix A describes how these are used in a typical scenario of DCE service location and invocation.

3.6 DCE Mechanisms and THETA

Aside from the use of DCE's Pthreads-based threads, little of DCE could be considered for use in THETA because of fundamental differences in message authentication and object location. It would not be possible for THETA to adopt DCE's approach to object naming and location, without foregoing the ability to enforce a mandatory policy on information flows resulting from messages between parties of potentially different mandatory attributes.

However, there are some possibilities for THETA/DCE interoperation and/or THETA use of DCE mechanisms in a limited way. Each of the following subsections provides a brief sketch of one possibility and some of its key issues.

3.6.1 Kerberos User Authentication

One possible use of DCE technology in THETA is in the area of authentication. DCE has extended the Kerberos framework of cryptographic authentication and service tickets to include the notion of a more complex credential which identifies a user *and* vouches for additional security-relevant user data, such as group membership.

One new approach to THETA authentication would be to "Kerberize" THETA as a whole, so that it uses a similar Kerberos-based approach to user authentication data in messages. Such a change would replace the current THETA approach in which user and group data is encoded in unprotected identifiers (THETA Principal and Access Group Set identifiers). The current approach requires the cryptographically secure THETA-kernel-to-THETA-kernel communication provided by TNET, as part of a chain of authenticity from a server back to a remote client: the COS allows local servers to have assurance of the authenticity of messages from the local THETA server; TNET provides one THETA kernel with assurance of the authenticity of messages from a remote THETA kernel; on such a remote host, the COS allows the THETA kernel to have assurance of the identity of a client.

In a Kerberos-based approach, this chain of authenticity is replaced by the scheme in which servers assurance of client identity is derived from the cryptographic credentials enclosed in messages from the client. Thus, in a Kerberized THETA system, the THETA kernel would still perform its required message mediation function, but would not be trusted to convey authentication data. Rather, the THETA kernel would simply forward successfully mediated messages with the sender's credentials intact.

The main new security issue in a Kerberized THETA would be the method by which THETA processes obtain their credentials. DCE processes directly contact the security server to engage in a dialog that results in the generation of credentials. Such direct dialog would be problematic in THETA, for two reasons: first, THETA's communication architecture requires the THETA kernel to be the mediator of all communication; second, because of the non-MLS nature of the existing DCE security server, and the multiplicity of sensitivity levels of THETA processes requiring credentials. To cope with this situation, the intermediary role of the THETA kernel would have be applied to communication between THETA processes and the security server, in a manner which would be invisible to the security server. Essentially, the THETA kernel (or some new, closely allied component) would have to act as a proxy for THETA processes on its host, and use DCE security server client stub on behalf of these processes; and it would have to perform downgrade and upgrade of information passing between THETA clients and the security server.

Finally, note that even in a Kerberized THETA system, there would still be a requirement for secure communication between THETA kernels. When a THETA kernel receives a forwarded message from a remote THETA kernel, the message meta-data includes information that the receiving THETA kernel needs to perform proper message mediation. This data must be securely transmitted. To meet this security requirement, security could be provided either by the current TNET approach, or by using Kerberos authentication on messages between THETA kernels.

3.6.2 Kerberos Kernel Authentication

A less ambitious use of Kerberos would be restricted to the area of communication between THETA kernels. Part of the above approach is securing this communication. One approach to doing so is embedding DCE credentials in messages between THETA kernels, rather than the current TNET approach. This change could be made without Kerberizing THETA as a whole, or could be done as an exploratory first step.

To use such a scheme, there would have to be a DCE security server (or a set of them) that would be accessible by every THETA kernel to obtain credentials for itself. No MLS extensions to a DCE security server would be required, but the server would become part of the TCB, being relied upon to correctly facilitate THETA kernel's authentication of one another.

However, this approach would not provide privacy and integrity of THETA kernel network messages. To meet this requirement, the THETA kernels could use a similar mechanism used by DCE RPC: make secrecy or integrity use of the session key enclosed in the Kerberos credential. Therefore, in addition to embedding DCE credentials in THETA Kernel Service Protocol (KSP) messages, the KSP would be also be extended to encrypt the payload (or just a one-way hash of it, for integrity only). Alternatively, the THETA kernel could use DCE RPC itself, leaving the KSP alone, but transporting KSP messages within DCE secure RPCs messages rather than the current approach of transporting them on TCP connections.

Note that the THETA kernel can directly contact the DCE security server just as a normal DCE client could, without the security problem described above for THETA processes directly contacting the security server. Unlike an arbitrary THETA process which could have a single sensitivity label not matching that at which the security server is running, the THETA kernel operates at a range of labels which, to facilitate interoperability, would be set to include the level of the security server.

3.6.3 THETA Client DCE Interoperation

Some amount of interoperability might be feasible between DCE and THETA. It is certainly possible for programs running on THETA hosts to call DCE stub code as well as THETA PSL calls.

The main issue to be dealt with is that such clients would be directly communicating with DCE servers. Clearly, mandatory controls on information flow would be required, very much in the manner that MLS systems currently interact with system-high-mode systems. One such approach assumes that all DCE services are on separate, non-MLS hosts that are accessible via network protocols to THETA processes. O/S-level controls on network protocol use would ensure that single-level THETA process may only use network protocols to communicate with single-level hosts with the same level as the THETA process. Depending on the COS, it may be that some amount of THETA involvement would be necessary to enforce such mandatory restrictions. If so, THETA would be involved to implement a mechanism very similar to that currently used for large messages. Thus, this approach uses MLS COS mechanisms to permit DCE interoperability without significant changes to THETA itself.

Even with this approach there are still issues of coordination of identity. A THETA process may well be able to use a PSL call to gain service of a THETA manager, and to call a DCE RPC to gain service of a DCE server. However, in the first case, the process's identity would be described by a principal object of the THETA authentication manager, which in the second case it would be described by an entry in the database of the DCE security server. The THETA manager would base access control decisions on the principle UID, while the DCE server would base access control decisions on the DCE credential issued by the security server.

In order for a uniform security policy to be enforced on these clients, there would have to be coordination of the authentication databases of the THETA and DCE portions of a system. Administratively, a THETA cluster would be embedded in a DCE cell, and the user identities and groups defined in the THETA Principle and Group Managers would be the same as in the DCE Security Server. Technologically, the THETA Principle and Group Managers could be privileged clients of the Security Server, in order to extract and copy back information from the DCE database to the THETA database. Clearly, there would be many details to work out, but the basic approach may well be technically feasible.

3.6.4 THETA Server DCE Interoperation

The previous section addressed the ability of programs running on a THETA host (and potentially acting as THETA clients or servers) to work with DCE servers on other hosts, by using DCE client stub software. By a similar arrangement, a THETA manager could also act as a DCE server by receiving DCE RPC messages for RPCs which correspond to operations on the type managed by the manager. As with the THETA client, the manager would be using network protocols to communicate with processes on remote non-MLS hosts, and the MLS COS must enforce suitable access controls on the use of the network.

Unlike client interoperation, however, meaningful multi-level service can be provided. THETA clients, being single-level, could only communicate with DCE servers of the same level. A THETA MLS manager, however, could be permitted to accept connections from DCE clients at any level within the manager's range.

Again, authentication and authorization are the central issues. If a THETA manager incorporated DCE server RPC stubs, then it would need to interpret DCE credentials, and map the information in them to the THETA authentication information required for THETA authorization checks. The THETA authentication database would have to be synchronized with the databases of any security server used by any DCE client that contacts a THETA manager.

3.6.5 Interoperation vs. Integration

With the approaches described in the above two sections, THETA/DCE interoperation would provide the full amount of THETA client/server functionality (with the exception of write-up operations) by means of eliminating the need for most THETA kernel functionality. The THETA kernel's location function for THETA clients of DCE services is replaced by client use of DCE's location mechanism. The THETA kernel's location function for DCE clients of THETA services is replaced by the THETA manager advertising itself (and perhaps its objects) in the DCE name space. The THETA kernel's message mediation function is not needed because of separate MLS controls on client/server communication.

Stated in this way, it almost sounds as though little of the THETA kernel functionality is required. However, this is not so. The relative lack of involvement of the THETA kernel stems from the basic nature of DCE interoperation as interoperation with processes on non-MLS, system-high mode untrusted systems. In order for the features of an MLS system– and particularly a heterogeneous group of them– to be harnessed, THETA is the necessary trust technology. The techniques discussed in previous sections are really general techniques applicable to any MLS system with requirements for local processes to interoperate via a network with remote processes on non-MLS hosts. The applicability to THETA of these techniques simply demonstrates that the benefits of a trusted MLS client/server distributed system can be combined with controlled interoperation with untrusted distributed client/server software.

The benefits of such limited interoperation— benefits including small impact on the existing THETA system— should not obscure the difference between interoperation and integration. It might be reasonable to integrate THETA and DCE, but this would be a much greater technical undertaking, with a different purpose: applying THETA technology to the implementation of DCE on a MLS system. Note also that such a project would still not obviate the need for DCE interoperability with non-MLS hosts operating in system-high mode, as discussed above.

3.6.6 Integration Approaches

There are two basic architectures to integration of DCE and THETA. The two architectures stem from two approaches to the central security problem: DCE allows arbitrary client/server communication without regard for information flow restrictions. One approach is to route through a new component the setup of all client-server connections, in a manner similar to the THETA kernel's mediation of large message facilities. COS security mechanisms would be needed to prevent clients and servers from making direct use of network protocols to form direct connections. This approach has a more THETA-like architecture, and would require modification or redesign of existing DCE components related to the current DCE RPC server.

To avoid such redesign, the second architecture permits clients to initiate connections to servers without mediation of a THETA-kernel-like component. Rather, COS security mechanisms would be needed to mediate the establishment of connections to servers.[3] As a result, DCE RPC software would continue to use COS services for networking, and implementations would not have to be changed unless the MLS COS interface required it. Thus, the MLS COS must provide some functionality for limiting or mediating access to networking, e.g. a MLS TCP/IP implementation. Furthermore, the feasibility of this approach may be effected by the heterogeneity requirement, for these mandatory access controls on network protocol interface to be enforced by a group hosts which may be running different MLS O/Ss.

DCE server development could also be used. Used as is, the DCE IDL and related tools would only be capable of supporting the implementation of single-level managers without a coherent ACL mechanism. However, the DCE IDL and related tools could be extended with THETA technology (or alternative THETA-influenced versions of these DCE tools could be provided) to support generation of MSL and MLS DCE servers. The DCE IDL could be augmented with notations for security attributes (e.g., read and write interfaces, new access modes). Rather then simply generating server stubs, the IDL tools could generate a whole server skeleton which uses the stubs to implement MLS or MSL functionality as in THETA. The manager skeleton would also allow all THETA/DCE managers to have the same ACL checking functionality.

---

[3] In this context, connection establishment would be either connecting to a TCP port, or sending a UDP datagram. That is, each datagram would be mediated.

3.7 Summary Comparison

Although THETA and DCE share many fundamental principles and architectural features, there are four very fundamental differences.

1. The mechanisms for message authentication are quite different, DCE's relying on a whole Kerberos-based security service and infrastructure, THETA's relying on plain-text authenticators protected by a simple end-to-end encryption mechanism.
2. The communication models are fundamentally different, with THETA relying on a message switch to implement message communication mediation, and with DCE allowing direct unmediated client-server communication.
3. The object models are very different, with THETA having an object model that is integral with access control mechanisms, while DCE does not include an object model.
4. Security functions are optional in DCE servers, though of course servers can be written which rigorously use them.

As a result of these differences, there is limited scope for use of DCE mechanisms in THETA, although there are possibilities for limited interoperation, i.e. access to DCE services from a THETA system, and access of THETA services by DCE. However, closer integration of THETA and DCE requires either significant changes to existing DCE components, or a solution to the problem of access control on network connections between heterogeneous MLS systems.

There are four areas in which THETA offers a superior approach to DCE.

- THETA provide high assurance MLS protection for untrusted clients and servers, something that is not possible with the current DCE architecture.
- THETA IDL tools produce an entire server skeleton to which only operation processing code needs to be added to fill in the stubs. DCE IDL tools also produces server RPC stubs that need to be filled in, but developers must write the structure of the whole server in which the stubs are embedded.
- The THETA manager skeleton and accompanying library code allow easy reuse of object access control functionality. Access control information is simply specified in the THETA IDL, and no additional software need be developed. DCE server's ACL management only partly matches the access control functionality of THETA servers.
- The THETA manager skeleton and accompanying library code allow easy reuse of a flexible replication mechanism. Replication information is simply specified in the THETA IDL, and no additional software need be developed. Reusable object replication functionality is not part of reusable DCE server code. Developing a replicated server (as in the CDS and Security Server) and developing a server with replicated objects (as with DFS) has been an *ad hoc* process in DCE, requiring significant technical effort.

THETA techniques in these areas would make valuable additions to DCE server technology.

4 Accomplishments

The THETA project has brought networking technology, advanced operating system concepts, and advanced security theory together with security engineering practice. This chapter presents our main successes in pursuit of this ambition. Problems, lessons learned (from both successes and failures), and future tasks are presented in following chapters.

In previous phases of the project, we achieved the following list:

- We developed a prototype multilevel secure distributed operating system for research, demonstration, and evaluation.
- ORA joined the Object Management Group (OMG) and worked on a security framework for CORBA.
- We designed and implemented an architecture that feasibly incorporates multilevel security into a contemporary distributed operating system (Cronus) without substantial loss of functionality or efficiency. The architecture also makes extensive use of the security controls of the underlying operating system thus avoiding duplication in THETA.
- We combined traditional and experimental engineering methods in the development of the system with some good results.

In the most recent phase of the project, our accomplishments include:

- We redesigned and formally modelled the kernel.
- We improved the usability of the THETA system for the developer, administrator, and user.
- We extended the specification grammar and enhanced the generated code for multilevel manager development.
- We completed an assurance guideline for CORBA ORBs and a vulnerability analysis guide based on our THETA experience.
- We achieved THETA technology transfer to the commercial world through our CORBA work.

The following sections describe each of the above listed accomplishments.

4.1 Secure Distributed Systems

Among secure networked systems, THETA is unusual in that it is also a secure *distributed* operating system. It differs from other secure networks in that security is addressed not just for interactions between network nodes, but also for applications that span many nodes. These applications may not even recognize that they are using distributed network resources, their security depends on global security properties of THETA.

THETA's security is distributed between layers of communications protocols. Networks usually address security at a relatively low protocol layer, like secure sockets and ports. In THETA, security is addressed at the higher protocol layers. THETA assumes security properties of low protocol layers in order to assure security properties of higher layers. Together, THETA is able to maintain secure communications without duplicating the services of the network.

4.2 Architecture

THETA's distributed, object-oriented architecture is taken without essential change from the Cronus distributed operating system. THETA's design is therefore an example that security can be incorporated into that architecture without sacrificing its advantages. Those advantages include allowing uniform access to resources of heterogeneous operating systems; organizing resources according to a type hierarchy, with inheritance of one type's operations by another; and extensibility of the operating system by addition of new types and managers for them. For details on THETA's architecture, see Section 1.4.

The key security problems solved in the THETA design were developing a secure protocol for supporting distributed systems; relating THETA security controls to local COS security controls for each kind of COS; minimizing the trusted kernel complexity; and retrofitting security to existing Cronus managers by building it into the manager autogeneration process.

4.3 Security Engineering

THETA's approach to security engineering has combined traditional, conservative methods with more experimental practices. The THETA kernel development emphasized the traditional approach. The functionality of the Cronus kernel was maintained, but the implementation was radically modified to implement mandatory access control and to meet the minimization requirement of the TCSEC. The THETA manager development focuses on trusted extensibility, which is the idea that THETA may be developed and adapted to new applications by adding new trusted and untrusted software.

A major THETA advance is the manager autogeneration process which simplifies the assurance arguments of new trusted managers. We have designed the THETA managers so they can

be run either as untrusted, single level managers or as trusted multilevel managers. Depending on the level of risk that is acceptable in a given environment, the managers may be instantiated appropriately.

For a more detailed discussion of the security features of the THETA system, see Section 1.5, in particular Section 1.5.5.

4.4 Redesigned Kernel

At the beginning of this project, our major goals were to incorporate multilevel security into an existing system (Cronus). At the time, we had targeted a single secure platform (AT&T System V MLS). The target platform severely limited our design. We have since moved onto other secure platforms (e.g. Sun Trusted Solaris and HP-UX BLS) which has permitted us to revisit our previous design decisions. The current effort included redesigning the THETA kernel. The primary goal of this redesign was to improve portability of the software, but, the new design also provides for higher assurance of security.

The new design addresses portability in several ways. First, since some of our potential target architectures have limited space in their process tables, we reduced the number of necessary kernel processes. Next, we was also improved portability by encapsulating operating system specific calls, such as interprocess communications, within modules. Finally, reliance on some details of COS identification and authentication was reduced by introducing a THETA login mechanism.

Experience with the prototype kernel has shown us which aspects of kernel functionality are fundamentally multilevel. As a result, the new kernel design has better segregation of the code responsible for MAC enforcement, which must be privileged with respect to the COS. This TCB minimization results in a clearer, easier security assurance argument.

4.5 General Improvements

The latest phase of the project has concentrated on making THETA more usable for all users involved, namely the developer, the site administrator, the THETA system administrator, and the end-user. The main improvements include restructured source code, better configuration management tools, increased and improved administration tools, expanded and improved documentation on all aspects of THETA, and concise configuration files to permit flexible, yet secure, management of the system. The biggest administration improvement has been the addition of a GUI interface (called Dream) which provides the administrator with the ability to view the status of the distributed managers/clients and perform some management of THETA entities.

Restructured Source Code. The layout of the source code is more intuitive and follows de facto conventions of UNIX applications. The source directory tree has "flattened" out. Previ-

ously, many directories had files that were links to or copies of other files in other directories, which caused several problems when installing and updating. These "cross directory" references have been removed by combining all logically related files into a single directory. In instances where logically unrelated files were still referenced, we created general "export" directories as a repository for these miscellaneous files.

For example, initially, the top level of the THETA source tree has several empty directories like `etc` and `include`. When the code is compiled, these directories become populated with "exported" files that various portions of the code reference and share. Therefore, modifications to any shared file can be made in a single place.

Another major improvement during the code restructuring was the revamping of the Makefile hierarchy. Previously, Makefile dependencies were passed along the chain via command line parameters. If a Makefile along the chain had an improperly set internal parameter or the parameter was missing, the remainder of the compilation could become corrupted. The Makefiles are now very readable, well documented, and follow the format of a common template. Also, global changes can be made easily in the single, master top level Makefile, which is included in all other Makefiles. This master Makefile is different for each platform. The THETA distribution comes with a sample master Makefile, named `thconfig.mk`, for each supported hardware platform.

Configuration Management. The THETA source code is under a more rigorous configuration management system. The configuration management plan, which is described in detail in [36], makes use of several vendor supplied tools. They are the Revision Control System (RCS), Concurrent Versions System (CVS), and RAZOR.

RCS is a public domain software tool suite that implements version control and modification logging on a per file basis. It implements version control by maintaining a control file in which a developer can check-out a specific version of the file, modify it, and check-in the modified file as a new version, logging the modification with a descriptive note. RCS provides a comprehensive tool suite for maintaining revision control on a single file. However, it lacks support for entire directories of files. CVS provides the capability of version control on directories by using the RCS tool suite. The sole purpose of RCS in this configuration management plan is to support CVS.

CVS is a public domain tool that implements version control and logging based on a directory tree structure of files. CVS also maintains a history database, in which all of its commands, check-out, check-in, tag, etc., are logged.

RAZOR is a tool sold by Tower, Inc. It is a configuration management tool suite that implements issue tracking with source code version control. It contains the **Issues** tool with which developers create bug reports, reports on design flaws, or development notes.

The combination of these tools and the particular configuration management strategy defined in [36] provide a solid configuration management plan with which to identify, release, and patch versions of the THETA system while concurrently working on mainline development.

Administration Tools. The administrative tools have been greatly expanded to minimize the effort required by the system administrator for such common activities as installing THETA, adding a new manager, creating / resetting manager working directories, adding a new user, and bootstrapping manager databases. A future goal is to enhance the tool set even more.

Documentation. The documentation set for the THETA system has been greatly expanded and updated. The following list of documents are particularly useful and have been much improved from their previous versions.

- *Introduction to THETA* - This document describes the THETA system, its architecture and capabilities. It is intended to be a high-level overview.
- *Manager Developer's Tutorial* - This two volume document leads the programmer through the steps involved in developing, testing, debugging, and maintaining a manager. Volume II details the manager generation process.
- *Software User's Manual* (SUM) - This document describes how to interact with THETA in a consistent and coherent manner. `tropic` is the main application discussed in this document.
- *Computer System Operator's Manual* (CSOM) - This two part document defines the role of the THETA operator and describes administrative duties such as starting, maintaining, and stopping the THETA system; the second part is the installation guide for THETA, which describes the general concepts of setting up THETA on a host and then details the different steps needed on each supported platform.
- *Software Programmer's Manual* (SPM) - This three volume document is a reference for THETA programmers.
- *Software Design Documents* (SDD) - This collection of documents provides details on every software component of the THETA system down to the level of pseudo code.
- *Version Description Document* (VDD) - This document specifies the version number of THETA in terms of its functionality, platform availability, known problems, and future work planned.
- *Formal Security Policy Model* (FSPM) - This document contains the THETA security policy and states how the formal model clearly maps to the actual THETA implementation.
- *Descriptive Top Level Specification* (DTLS) - This document describes the various THETA components at a high level.
- *Trusted Computing Base Configuration Management Plan* (TCBCMP) - This document describes the configuration management plan for the trusted computing base

code. This configuration management plan is an important software engineering step that facilitates version identification.

Flexibility. Portability of the THETA manager and library code have been greatly increased. Within manager code and its support libraries, there were many directory paths and file names that were hard coded. These have since been substituted with utilities that allow flexible, relative paths. Also, formerly, some unique identifiers for certain key principals were hard coded into some managers. These have been removed where possible. The Authentication Manager still contains a hard link to the principal `theta` who is given the privilege of administrator within the THETA database realm.

Configuration Files. The THETA system relies on several configuration files. Previously, those files were interdependent, and modifications to these files often led to inconsistent installations. The dependencies between files have been removed, and the files have become much more understandable. Another vast improvement in the configuration files is that the data is now (mostly) in human-readable format, unlike the previous hex numbers.

Examples of configuration files, which are detailed in [29], are found under the top-level source directory in `src/conf/<operating_system>`. The currently supported platforms, HP-UX, HP_BLS, SUN_OS, and SUN_CMW, are the options to enter in place of `<operating_system>`. The configuration files found here are

- Low — THETA lowest COS level specification
- Levels — THETA levels and mapping to COS levels
- Categories — THETA categories and mapping to COS categories
- RangeCompartments — THETA range compartments
- Users — THETA principals, their attributes, and mapping to COS users
- Managers — THETA managers' specifications
- DefaultAuditEvents — Default auditing events
- Networks — THETA network specifications
- tnet.config — TNET configuration parameters
- Labels — For unlabeled OSs - name to sensitivity label mapping
- Clearances — For unlabeled OSs - user clearances

4.6 Manager Development

The Automatic Code Generation (ACG) manager has been enhanced to allow greater expression in the manager specifications. The developer can now attach security levels to various data types, data elements, and operations. The ACG manager is responsible for parsing the multilevel specifications and producing code that can safely handle multilevel data. In order to

assure safe handling of multilevel data, the generated code should have the following attributes.

- **Small size**, in terms of the number of lines of code, contributes an obvious benefit since this provides a reasonable bound to any certification effort. Since programs that manipulate data at various levels are necessarily part of the TCB, minimization of program size is a requirement for high B level certification.
- **Simple coding** makes software more readable, understandable, and verifiable. If the code is clear and concise, then it much easier for an analyst to identify security properties of the program and verify their correctness.
- **Layering** of software by building programs up from small, simple modules simplifies security analysis by allowing evaluators to follow a divide and conquer methodology. Each module can be evaluated separately and then the higher layer can be evaluated for its contribution to the program as well as its use of the modules. Once each module's security properties are determined, the analyst can concentrate on the higher layer without having to include the raw code from each module. Layering has the effect of making software, with a large global line count, small and simple when viewed layer by layer.

THETA MLS managers support objects and multilevel data over a range of security levels. Since such managers reside in a *single* process address space and cannot employ hardware mechanisms to enforce separation of data by security level. The guarded data structure method (GDSM) is provided to compensate for the lack of hardware enforcement. The GSDM method uses level factoring of data and information hiding to separate data and objects by security level, as well as to limit access to multilevel data structures. This method is described in much greater detail in [32].

4.7 OMG

Based on our CORBA and THETA efforts, we have identified a number of critical issues in secure distributed object systems. These issues include definition and maintenance of dynamic security policies across domains, security enforcement, security administration, and assurance. Many of the same issues we dealt with in THETA surfaced in our work in defining the security framework for CORBA. Our experience with THETA proved invaluable in discussing these issues with the security submitters group. As the only MLS ORB implementor, ORA was relied upon by the standards group to represent and defend the government security requirements.

Based on ORAs discussion of assurance issues, ORA developed an outline for an assurance criteria for ORBs. The vendors will be required to supply this product profile to convince potential users that security is enforced by their product.

5 Lessons Learned

A quick summary of the lessons learned is listed below.

- Mapping THETA's general security policies onto platform specific policies introduces several technical and administrative difficulties.
- In contrast to our initial view, we now view the THETA kernel as a relatively non-portable component.
- System maintenance for distributed systems inherently have some consistency problems, and these problems are compounded with heterogeneous platforms with diverse security policies.
- Better configuration management tools would ease some of the complex configuration management policies that we currently implement.
- Formal modelling of the new kernel design has provided several new insights.
- The replication protocol has proven to be rather complex in a multilevel environment, and it currently does not work in THETA version 2.2.
- The Trusted Networking (TNET) component has a design problem for which we offer some solutions.
- Several issues should be investigated before choosing a new platform on which to port THETA.
- As the first extensive user of the THETA system outside of ORA, TIS has provided several insights into the design and implementation of THETA.

5.1 Layering Security

There is sufficient difference between THETA and UNIX security policies to cause technical and administrative difficulties. THETA's security policy permits multilevel processes with restrictions on the range of levels; not all secure operating systems permit this such as Sun CMW. THETA permits degrees of trust in processes; some secure UNIX systems insist that a process run single level, or else the UNIX kernel foregoes all security checking by permitting the process to have unlimited access to all information in the system.

5.1.1 Mandatory Access Control

The main problem we had to solve in layering THETA mandatory security on UNIX was how to represent THETA processes with limited trust (restricted ranges) on a UNIX host that had no notion of such a thing. We chose to map each range of THETA levels onto some individual UNIX level.

Mapping of THETA level-ranges to UNIX levels places trust in specific integrity properties of the COS that the COS vendor may not have intended or given sufficient assurance; thus, our solution is not ideal. Despite this disadvantage, mapping THETA level-ranges to UNIX levels seemed to be the best approach for demonstrating range restrictions in the prototype. This demonstration gives a realistic simulation of a COS that *does* enforce range restrictions and thus gives the user an opportunity to experiment using THETA on a such a system. However, this approach is acceptable only for demonstration purposes. For a fielded THETA system, if the COS does *not* support range restrictions for processes, then THETA should not. That is, THETA should never be used to add new security features; it should be used only to extend security features to an object-oriented environment. A consequence of heterogeneity may be that the full generality and flexibility of THETA's security policy may not be available on some hosts if the COS does not have the right enforcement mechanisms to support a THETA abstraction.

5.1.2 Discretionary Access Control

Layering THETA discretionary security mechanisms is much easier since it does not require anything special of UNIX. Since each THETA manager has a separate UNIX identity, it is a trivial matter to set up THETA managers so they are the UNIX owner of all object database files that they manage. The UNIX protections can be set so that only the owner (manager) has any access to the object databases containing relevant THETA objects. This ensures that the manager mediates all accesses to the THETA objects since there is no direct access via UNIX by unauthorized parties.

5.2 Portability

THETA Kernel Portability. When the development effort began, THETA's design was strongly influenced by the target platform. Restrictions imposed by the operating system produced a somewhat awkward implementation. In the current phase of the project, we had the opportunity to rethink the design, especially with the goal of developing a portable kernel. We have improved the kernel to lessen the burden of the porting process by encapsulating low-level operating system dependencies as much as possible. Despite our redesign, the kernel porting effort remains higher than other THETA software components. The effort is necessarily larger than any other piece of THETA since the kernel is the software that must integrate the various capabilities of the local operating system. However, the extra time needed to port

the kernel eases the burden for the managers and clients, since the kernel abstracts the operating system differences from those components.

Manager Portability. In developing the Cronus object managers, BBN adopted a software development philosophy that has resulted in highly portable code. Inheriting those Cronus features, THETA object managers are also highly "self-contained" and rely little on software outside the manager itself. This aspect of managers, more than any other, provides substantial portability both in Cronus and in THETA.

The best example illustrating these ideas is the manager tasking package. Largely because of the manager tasking package, a THETA object manager is a complete operating system in its own right. Concurrency is implemented as set of communicating tasks: this approach is well-known and flexible.

Although Cronus and THETA managers share the features described so far, the manager tasking package developed for THETA is very different than that used in Cronus. In designing multilevel, multitasking facilities for THETA object managers, there was a choice: use the multitasking of an underlying COS, or simply build a complete multitasking facility. In THETA, we built our own multitasking facility which resulted in highly portable software since manager tasking facilities have negligible dependence on the underlying COS. If we had elected to implement multitasking directly on the COS, the power and portability of the manager tasking facilities would have a strong dependency on the strengths and weaknesses of the equivalent COS facilities—the portability would be easy or hard depending on the COS.

5.3 System Maintenance

As a distributed, heterogeneous system, THETA has a fairly complex configuration and it is difficult to administer due to its heterogeneous nature. Properly maintaining the configuration files for a particular installation is essential for the security of the system. We have extended the tools for the administrator to ensure that the security integrity remains intact throughout the migration of the system installation guide. See [29]. A future goal of THETA is to remove as much administrative complexity as possible and to provide tools to check for network-wide consistency.

5.4 Configuration Management

In the most recent phase of the contract, we have focused on configuration management of the source code as well as the system set up files. We have surveyed several configuration management tools, and we are actively looking for better ones. We are not completely satisfied with our current tools set, however, we have adapted our configuration policies to compensate for the problems with the tools.

We are using RCS to maintain updates to individual files, CVS to maintain updates to entire directory structures, and RAZOR to track and document the modifications and bug reports. Each individual tool does provide each desired function; however, no single tool is *good* at *all* of them. Therefore, we have adopted a configuration plan that utilizes the tool that is best for each particular function. This plan is further described in [36].

5.5 Formal Modelling of the Ada Kernel

A new THETA kernel was designed and implemented in Ada. A formal specification of the kernel using the Larch/Ada specification language was part of this effort. This section describes this work in detail. The C version of the THETA kernel remains the system production version.

The new design had three aims: to make THETA more portable, to make THETA easier to maintain, and to simplify and clarify the *assurance argument*—the combination of formal and informal arguments offered as evidence that the information protection mechanisms succeed. We are using formal techniques, principally formal specifications, to help achieve both portability and a high assurance of correctness. A key requirement for portability is a precise definition of the interfaces between the kernel and its applications, its host operating system, and the network.

Our goals for the formal specification of the kernel include the following: to clarify our own understanding of the design and implementation, to guide writing the documentation, to aid long-term maintenance (which includes the training of new implementors), to help define a security policy, and to provide the basis for an assurance argument which shows that the kernel implements the policy. A description of this work appears in a chapter of a book entitled *Applications of Formal Methods* [6].

5.5.1 Methods

No ready-made formalism existed that would easily accommodate all aspects of the kernel from its top-level view as a distributed system down to the Ada implementation of individual code modules. Thus, certain understandings about the function of the system would remain informal. Therefore, our final model contained both formal and informal arguments.

Consider a bottom-up view: At the bottom of the system are modules of Ada code. To represent their integration into the distributed THETA kernel, one must go outside the semantic model of Ada, which defines the meaning of a single program, not of a collection of programs that cooperate. We decided that the best available techniques for specifying and reasoning about Ada code were the Larch/Ada specification language and the Penelope system for formally verifying Ada code [5]. These techniques apply to a subset of sequential Ada. The challenge was to map specifications of properties of distributed systems into Ada code.

Now consider the view from the top down: The end-user sees a system that offers a set of transparent services and applications. The fact that the system is distributed is hidden from the user. One step below that is the level that concerns us, a more concrete view in which the system is manifestly distributed (and therefore concurrent). That is the level at which the kernel appears. The challenge here is to map specifications at the user layer to the distributed OS layer, and to the kernel layer.

The kernel's message routing involves elaborate protocols that, among other things, search through the system for migratory objects. The mechanics of the kernel-to-kernel protocols can be naturally described in a process algebra semantics such as LOTOS [2,3], which was devised specifically for such purposes. We decided, that a process algebra specification would involve too much detail and detail of the wrong kind. For example, the details of the "locator" protocols are largely irrelevant to understanding the system-wide properties we considered. The strategy actually used to locate recipients—broadcast queries, consulting caches of previous replies, etc.—is largely ad hoc, arrived at by experimental tinkering to improve performance. For our purposes the important point was not to describe or predict the precise route of a message or the precise way in which that route would be chosen, but rather to describe the invariants that any satisfactory route must maintain—invariants such as "the host on the path of each leg of the journey must have an appropriate security range." But the notion of state is not directly or conveniently expressible in typical process algebras.

Our needs suggested a state machine model in which we could express invariants on states and on histories (sequences of state transitions). That still leaves open the question of how best to express the desired invariants. We could explicitly construct the domains of states, transitions, and histories, and use ordinary logic to define the predicates of interest. Alternatively, we could represent the model in a temporal logic, where states are defined explicitly and histories are present implicitly in the semantics of the temporal operators. Our previous experience suggested that temporal logic would not be appropriate, since it seemed better suited for describing the behavior of reactive systems—behavior such as "Every time A happens, the system must do B before the next C"—than the kinds of invariants we wished to enforce.

These considerations, justified the following plan: A simple formalism expressed in terms of state machines and ordinary logic would be suitable for the most abstract representation of the system. That model could be expressed straightforwardly in many of the formal notations such as Larch Shared Language (LSL), Z, or PVS. Larch/Ada, described below, seemed the obvious choice for describing the code modules themselves, though it could not fully describe modules whose behavior is essentially concurrent. The Penelope system could be used to develop both the abstract state machine description in LS, and the specification of code modules in Larch/Ada. Penelope would also support formal verification of the code. Whatever hierarchy of refinements we provided between these two pictures would have to contain a gap, bridged by informal explanations gluing the distributed modules together outside the semantic model of Penelope and Ada.

We concluded that we should use our own collection of Ada methods and tools—particularly since we wanted to leave open the possibility of formally verifying modules of the code. There seemed to be no theoretical grounds for thinking that any other methods would be superior and the practical advantages would be considerable: In house experts would be available and training would either be unnecessary or easily arranged. It would also be possible to ask for (small) modifications of the tools to meet our needs.

5.5.2 Formal Methods Tools

We wrote specifications in Larch [7, 8, 9] (a notation for first-order logic) and Larch/Ada [18, 19] (a formal specification language for sequential Ada). The code, specification, and English-language documentation are maintained in noweb [25, 27]. We used the Penelope system [5] to check the syntax and static semantics of the specifications—and, to prove a limited number of properties of the specifications and to prove the correctness of the corresponding Ada code.

We applied other off-the-shelf formal techniques opportunistically, whenever it seemed convenient to do so. We built a simple model of THETA in Romulus [20], which can be used to check that components of the design satisfy a particular composable security property called restrictiveness [12, 13]. We also checked the Ada code with AdaWise [1], which automates tests for certain common dynamic semantic errors.

These methods and tools are briefly described below.

Larch. Formally, the Larch Shared Language (LSL) is a notation for describing axiomatic theories in a well-organized, modular way. Its great virtue is simplicity: An LSL *trait* defines a theory of ordinary first-order logic, together with certain assertions about that theory, such as the assertion that it implies some other theory. The structuring constructs are few and almost self-explanatory. We have found it convenient to extend Larch somewhat, and in what follows "Larch" and "LSL" refer to our extended language.

Larch is a general methodological program that has two parts: developing a reusable library of mathematics (expressed in LSL) that is independent of any programming language, and developing specification notations, called *interface languages*, that are tailored to particular programming languages. An interface language provides a way to specify a program by relating its behavior to the behavior of abstractions defined in LSL. For example, we can define the abstract notion of an unbounded stack in an LSL trait and use that notion to specify a bounded stack package in Ada along lines suggested by the following informal description of "push": if the stack is full, the executable push operation raises an exception; otherwise, pushing updates the stack as does the mathematical push operation defined in LSL. The same LSL trait could be used to specify a push operation in C, where an attempt to push an element onto a full stack is signalled not by raising an exception, but by returning an appropriate integer status code.

This style of specification is called "two-tiered" specification. The specification of a particular program module consists of a *mathematical tier* (a theory defined in LSL) and an *interface tier* (annotations defined in the appropriate shared language).

Two-tiered specifications pay an extra dividend on THETA: We have written very little Larch/Ada and a great deal of LSL, and we expect to reuse the LSL for specifying the kernel's implementation in C. (There is an interface language for C, called LCL [9].)

Larch/Ada. Larch/Ada is an interface language for Ada. Our specifications used features of Larch/Ada that are in draft form and not currently supported by Penelope.

Penelope. Penelope [5] is an interactive tool that permits the incremental development of Larch/Ada specifications, Ada code implementing those specifications, and machine-checked proofs that the code is correct. The underlying formal model of Ada semantics covers virtually all of the machine-independent features of sequential Ada. Penelope currently implements a subset of that model, which includes generics, packages and private types, and exception-raising and -handling.

AdaWise. AdaWise can be regarded as a kind of super-lint for Ada programs. It checks, for example, whether a program depends on the order in which its constituent modules are elaborated (roughly speaking, initialized) at program start-up. That order is implementation-dependent. Penelope does not make AdaWise redundant AdaWise accepts arbitrary Ada programs and does its checks automatically; though the corresponding checks in Penelope, when applicable, are more powerful.

noweb. noweb [27] is a simplified variant of Knuth's Web [10], which supports "literate programming"—documentation that organizes the description of specification fragments and code fragments in the most informative expository order. We maintain a single marked-up set of source files containing specification, English-language commentary, and Ada. The noweb tools can "weave" these source files to produce `LaTeX` source for printing documentation organized for readability and can "tangle" these source files to produce code and specification in syntactically and semantically legal form.

We have successfully used a version of Web to maintain the code for the Penelope system through several years of implementation and many changes in the team of implementors [26].

Romulus. Romulus allows a user to model a system as a collection of interconnected processes, to show that the components are secure, and then to apply a somewhat sophisticated theory of composable security properties to show that the overall system is secure.

5.5.3 Evaluation and Conclusions

The portable kernel is currently incomplete since it does not implement all the kernel's functions. It has been successfully tested on a network of two machines, one of them running the existing complete THETA kernel, and the other running the partial portable kernel and pro-

cesses that simulate the actions of managers and clients. Our lessons learned from this experience are in the process and the supporting tools areas.

Process. Due to time and resource constraints, writing the formal specification did not contribute as much to creating the design as had been hoped. With a better understanding of what the specification effort should provide, we could have utilized the results more efficiently. Our formal methods experiment started from scratch, so that the experience base developed—understanding the application domain, developing libraries of specification concepts, etc. will provide a useful base for the future. We are past the initial learning curve, and we feel that future formal methods experiments will be far more useful and effective due to out past experience.

Tools. The tools that we used were adequate in their own right, but did not make up a coherent development environment. Since, in particular, some of them were predominantly batch-processors and some (to varying degrees) incremental, it would have been difficult ever to make them work together well. Similar difficulties arise in integrating noweb with an Ada debugger or an Ada library system. (The reason for integrating with the Ada library is to avoid unnecessary recompilations of library units.)

The problem with tools is both pragmatic and theoretical. The immediate pragmatic problem for a user of formal methods is to choose some technique whose tools can be made to work reasonably well with the rest of a development environment that is already in place and cannot be radically changed. The more theoretical problem is that one often wishes to apply different specification techniques to different aspects of a problem or to different levels of abstraction, and there is little support for this activity, either conceptual or automated.

Conclusion. We used these methods to improve the quality of our system, satisfy our customer about that quality, and help with the long-term maintenance of the kernel. We also hope that the experience acquired by doing the formal work will be useful in future work on distributed systems.

The main benefit to date has been a well-organized understanding of the kernel and its role. In particular, we have a standardized and rationalized vocabulary for describing the kernel and the security properties we want to guarantee. We have good reason to attribute much of that success to the formal work.

5.6 Replication Complications

During our extensive modifications to THETA, the replication capability was given a lower priority in comparison to the goal of encapsulation and usability. In the midst of massive overhauls of the system, we have disturbed the delicate balance in the replication protocols. We had some problems with replication on large datatypes before, however now, replication does not work reliably on even the smallest of objects.

Under our current code configuration, replication worked once under very particular conditions, which are minimal CPU usage on the involved machines, very small object size (i.e., UID only, no other data), the machine running the manager with the original database has the debug flag on (to slow it down on purpose), and the database to be replicated is the generic only. Replication also works usually if both managers are started with the "/nodac" option.

From the series of tests performed, we believe that there is a timing problem and possibly a memory problem. Since replication works when discretionary access control checks are turned off, we investigated that portion of the code in detail.

While processing a replication request, it appears that the manager is denied access to its own databases. A timing problem occasionally causes a null access group set to be returned during a regular DAC check, thus the manager denies itself access to its databases. The symptoms are fickle and difficult to duplicate exactly, thus this problem has been tough to identify more specifically.

Another possible timing problem may occur when the host holding the original database sends the response sooner than the receiver is expecting. Thus the receiving process drops the premature response, then wakes up, too late, and times out, and the replication request fails.

5.7 TNET Experiences

This section specifies certain aspects of the Trusted NETwork (TNET) communication service as delivered by Trusted Information Systems (TIS). We also list a few bugs that were fixed by ORA, as well as some areas to consider redesigning for improved correctness and efficiency.

5.7.1 TNET Overview

TNET is designed and implemented as a sublayer of the THETA kernel. The TNET sublayer provides a trusted communications service that protects all data transmitted between pairs of THETA kernels connected to the LAN. TNET does not protect communications between individual clients and managers and the THETA kernel.

The TNET design incorporates cryptographic-based security services directly within the THETA kernel. Four basic security services are provided.

- Data integrity: unauthorized modification (including insertion or deletion) of data sent across the LAN can be detected.
- Data confidentiality: data is protected from unauthorized disclosure.
- Source authentication: the destination THETA kernel is assured that it is communicating with a peer THETA kernel.

- Multilevel secure separation of message traffic: messages are assured to be at a sensitivity label within the range of security levels for which the THETA host is authorized.

TNET uses an encapsulation technique to apply data protection mechanisms (i.e., encryption and cryptographic checksum) to the THETA KSP messages. TNET provides these services on a THETA kernel-to-kernel basis using symmetric (i.e., secret) key technology which makes use of cryptographic keys and secret key cryptographic algorithms. The cryptographic keys are uniquely associated with one or more entities and must not be made public. The use of the term "secret" in this context does not imply a classification level, but rather implies the need to protect the keys from disclosure or substitution.

Sets of secret cryptographic keys (one per security level) are shared among the kernels connected via the LAN. Security labeling of the KSP messages is enforced by THETA and the underlying COS. The TNET communications service computes a cryptographic checksum for the message data and seals it using a secret key associated with the security label stored in the Kernel Reliable (KREL) or Kernel Service Protocol (KSP) Header. The protected THETA message is encapsulated with the TNET Header and passed to the COS Transport Layer protocols. Source authentication and multilevel separation of messages is provided by acquiring the key identifier associated with the security level of the KREL/KSP Header. Data integrity and confidentiality is provided through the use of the cryptographic library routines. The TNET header is described in detail in the Kernel SDD document [32].

5.7.2 TNET Enhancements

Performance. The current implementation can improve performance through more efficient data handling. In the TNET `tsend` function, the implementation disassembles the packet by copying most data into separate data structures and then filtering it through checksumming and encryption phases. The `tsend` call then reassembles the data in a new contiguous packet allocated from the heap, and then sends that new packet down the stream.

The `tsend` call can be made more efficient by taking the buffer apart, calling the encryption and checksumming functions, placing the converted data in static buffers, and then sending the parts down the stream in the proper order, without reconstructing the packet in contiguous memory.

Communication misconception. The implementation was based on a design that assumed data which was sent on stream channels worked much the same way as data sent and received on datagram channels. If a UNIX `send` call of a certain small buffer was sent, it was assumed to be received in its entirety. Between two exact same machines (SunOS on Sparc 2s) with an unloaded network, communication would synchronize in this fashion, due to the similarity of speed and implementation of the underlying operating systems and hardware. However, when using disparate architectures, e.g., AT&T PCs and SunOSs, this assumption was no longer correct. The implementation, based on this design, sends a TNET header and expects a complete message to be received on the other end. The receiving end would extract buffer length

information and, assuming that the next incoming buffer was being sent in a contiguous chunk, it would be received that way. This type of buffer framing is not guaranteed with communication on a loaded network; even the relatively small TNET header could not be guaranteed to be received as a whole. The result caused many communication "drops" at ORA, especially when using the HP BLS, a different, faster architecture, and caused the THETA system to ultimately fail.

ORA patched the implementation by generating a function, called `guaranteed_recv`, that polls the stream channel until the requested length of data was received. This effectively makes a non-blocking channel a blocking channel.

TNET and non-blocking communication. THETA was designed for stream communication to be non-blocking. TNET bases itself on packetized encryption, both in datagram and stream communication. Packetizing works properly for the datagram communication as long as the length encrypted and checksum packets are less than the maximum UDP packet size. For stream communication in THETA, however, messages must be received along the stream in their entirety before they can be decrypted. This violates the non-blocking requirement for the THETA communication, as the TNET `trecv` call must effectively read the entire packet possibly using multiple UNIX `recv` calls before giving control back to the caller. This problem effectively creates a blocking read on a non-blocking communication channel.

Key management: Encryption keys are kept in a file on each THETA system, and this file must be the same for all THETA implementations on the same network in order to communicate. Distribution and management of these keys becomes a physical problem. To ward against decryption analysis attacks, keys should be changed periodically. To change an encryption key, all THETA nodes must be shutdown, the key file redistributed by some secure method, and all THETA nodes rebooted with the new key file.

This distribution method is not practical and is difficult to administer. The limitation has been acknowledged by TIS. TIS chose this distribution method to expedite the development with the intent to change it to be more robust in future releases.

5.7.3 Solutions for Non-Blocking Communication

The problem that results from packetizing stream based communication hampers the THETA system enough that we must consider some alternative implementations. There are two possible solutions; however, both require a considerable amount of re-engineering. We will summarize both, but first, we will briefly discuss the current encryption model. The model for TNET was created with the purpose of installing a network encryption scheme into THETA without affecting THETA's operation. TNET was designed so that its interfaces modelled the interfaces for UNIX communication system calls. For example, TNET has `tsend` for UNIX send, `trecv` for UNIX `recv`, etc. The model is illustrated Figure 5-1. This model demonstrates that an encryption scheme, if tailored to the interfaces of UNIX, could be slipped in without much difficulty.

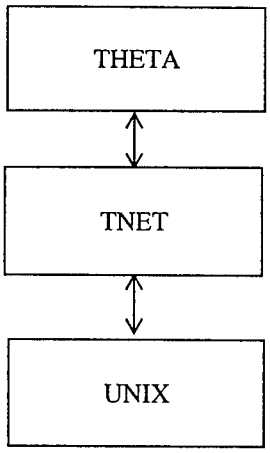

Figure 5-1: Current TNET Encryption Model

Solution 1 - New encryption model. The first approach is to change the encryption model to move TNET encryption off to the side, as shown in Figure 5-2. Communication would remain packet based and dependant on the structure of the KSP packet. This model permits end to end encryption to be easily retrofitted. A packet going out is specially encrypted by the KSP. The packet is then sent out on the usual THETA mechanisms that adhere to the non-blocking requirement for communication.

This solution involves modifying the `dirsendrecv` CSU which is relatively easy to do in the NetMessageOutAndStatus() function. This function can call an encryption and checksum routine, creating a new buffer to be sent over the network. However, the methods used for receiving the message, in NetMessageBodyInAndStatus(), break up the receiving of KSP packets from the network into two parts, first the KSP header is read, then the body of the packet read, based on different attributes in the previously read header. Since the entire packet would have to be received in order to be decrypted, this function must also be aware of the TNET header. Another option is to have NetMessageOutAndStatus() encrypt the KSP header and body separately. In that case, both functions would have to coordinate separate encryption and decrypting of headers and bodies. These modifications result in a considerable engineering effort.

Solution 2 - Same model, slightly different strategy. The second solution is to modify the current TNET implementation to not continuously poll the communication for any receives. This change can be done by having the TNET also implement the select() call differently for encrypted streams. This change would effectively treat incompletely received packets as "ready to read", thereby fooling the NetworkIsReady() call in the `dirsendrecv` CSU that there is data pending on the stream channel. However, to get this to work correctly, the subsequent `recv`() call must return an indication of 0 bytes read when it cannot construct the complete packet. This would require a change to the `diresendrecv` call, not to equate 0 bytes

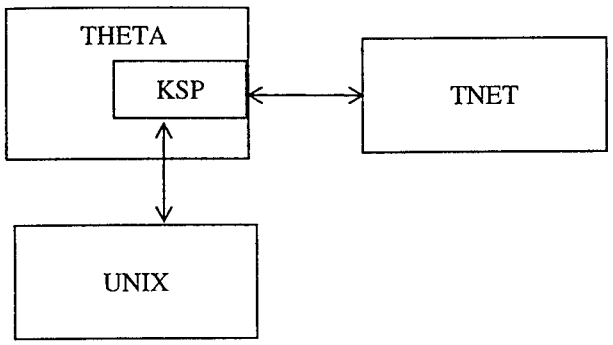

Figure 5-2: Alternate Encryption Model

read with an error. The THETA system is fooled into believing that there is a packet ready to be received, however it is not yet completely received. THETA will continue to poll the TNET `trecv` call until the packet is received. So, TNET will not return any incomplete packets, just complete decrypted KSP packets, or a zero bytes read status. This effort would require modifications to the `dirsendrecv` process, only on the receiving side, and some modification to the TNET code in order to "remember" the incomplete packet between TNET `trecv` calls.

5.7.4 Summary

Solution 1 is preferable. It has the drawback of modifying existing THETA code to be encryption aware. However, modifications to make THETA encryption aware, at least for packet style communication, should be of a standard nature so that other encryption systems may be easily retrofitted in TNET's place.

Solution 2 is probably the quickest solution for the short term; however, the ramifications of creating a TNET select call for NetworkIsReady(), or possibly changing the entire logic of the receive process must be investigated.

5.8 Porting Experiences

We have had some difficult experiences in a few of porting attempts which may have been prevented had we had more foresight. We have learned that much more investigation should go into selecting new platforms to support THETA. For example, available operating system resources and THETA's resource requirements should be well understood and clearly stated when choosing potential platforms. Also, proprietary claims of the commercial vendors should be thoroughly investigated before making a final decision to port to a particular platform. Our strategy was to initially run THETA on the AT&T machine. The development environment of AT&T was so primitive that we ported the development to SunOS and did most of

the work there. Upon completion of major milestones, we moved the code to the AT&T for testing. Once the AT&T port was complete, we then ported to the CMW and HP. From the CMW, we did a fairly straightforward port to the Trusted Solaris platform.

5.8.1 Trusted Xenix Experiences

The port to Trusted Xenix was hampered by lack of sufficient operating system functionality that was taken for granted by the THETA system. For example, socket calls required by THETA communication were not supported by the Trusted Xenix operating system; thus, the necessary functionality needed to be added to Trusted Xenix. Tailoring communication mechanisms quickly became very expensive and time consuming.

A second major problem was mismatches in very low level system features. For example, there were differences between THETA and Trusted Xenix's word sizes and macro sizes. Also, THETA had large memory requirements which needed to be mapped into Trusted Xenix's memory model. The frustration associated with these issues was that they consumed much more time than anticipated, but were vital to the completion of the port.

Trusted Xenix was an old PC based operating system and did not provide sufficient services necessary to run THETA. Furthermore, Trusted Xenix lacked the necessary development tools to perform this task. Finally, TIS began the port with an older version of THETA. These problems resulted in an incomplete porting effort of THETA to the Trusted Xenix platform

5.8.2 Lock Experiences

Proprietary claims on hardware, software, training materials and documentation were asserted by Secure Computing Corporation (SCC) after their equipment was purchased. As a result of those claims, our subcontractor, Trusted Information Systems, was not permitted access to necessary information and equipment in order to be trained on SCC's Lock system. After this problem surfaced, the government made the decision not to port THETA to the Lock system. Being made aware of these proprietary claims in advance would have prevented this unfortunate outcome.

5.9 Application Development Comments

TIS was the first extensive users and developers of applications for THETA outside of ORA. The following section provides useful experience and feedback on the design and implementation of THETA. ORA had continuous contact with TIS during their development process; therefore, we were able to address several of their modification suggestions when we restructured the system. Rome Lab, NRaD and CECOM also installed the system. We will note the areas where particular problems have been addressed and note others that are still unresolved.

Manager Generation Mechanism. The manager autogeneration mechanism worked very well, and implementation of simple THETA managers was fairly easy. The type definition and manager specification languages are easy to understand and use and well-documented. The manager development process is extremely well structured, in that the whole framework is automatically generated and only the operation stubs need to be filled in. The manager code examples from ORA were especially helpful. Once one type and an operation had been defined and implemented, the definition and implementation of additional types and operations was very straightforward.

THETA Installation. There was very little documentation for THETA installation and application development; existing documentation was either incomplete or out-of-date. This was improved in the latest version of THETA. The installation process is documented in [29] and several labor-savings utilities have been created for the administrator and the application developer to ease the process of maintaining the system. Although these were necessary improvements, Rome Lab felt that they still needed to be very familiar with the underlying OS and THETA. They annotated the manual with additional notes before sending it to NRaD and CECOM. Clearly, more needs to be done in this area.

Administration Complexities. The administration of a THETA system is non-trivial and requires improved diagnostic tools. As stated above, THETA has vastly increased and improved the tools for installing, maintaining, monitoring, and modifying the THETA system. Error reporting of these new tools still needs to be improved.

Debugging: THETA needs better debugging tools to improve the application manager development process.

Manual Alteration of Generated Code: The Regrade Manager is a very simple manager, the sole interesting feature of which is the use of the General Purpose Demon (GPD) feature of the THETA manager skeleton. The regrade manager's GPD code calls a PSL routine of another manager. However, the normal autogenerated PSL did not function correctly within the GPD and had to be re-written. Ideally, PSL calls should work the same in GPD code as in the main body of manager code, without needing to be rewritten.

Lack of Manager Support for the Unix System(3) Call: Since the dBase IV DBMS (DB4) did not provide a programmatic interface, rather it only provided a user-oriented interface, it was necessary to invoke DB4 from within a THETA manager using Unix shell commands. However, THETA manager implementation had side-effects on the Unix interface, so that the system(3) library routine did not function properly. The programming workaround of using fork-and-exec was fairly simple to put in place.

Locator Bug: For reasons never clearly articulated, it was necessary to explicitly inform the THETA locator that system manager object resided locally. To do so, we used scripts of tropic locate commands that were routinely performed after starting the THETA kernel and the authen, process, and audit system managers. Without this step, THETA would sometimes not function correctly to support the demonstration application. The locator bug is still under investigation.

Registration Error: Occasionally, registering with the THETA kernel resulted in the following error on the CMW:

> Refer to Trusted Facility Manual Appendix to identify unknown error codes. THETA Process with UNIX ID 1481 exiting — can't create log file. KERR_Startup:fopen:: Permission denied

Attempting to register again will usually succeed. The failure is due to an attempt to create a log file which already exists (re-use problem). Kernel log files should use both the process id and date/time in constructing their names or some other scheme to avoid name conflicts. This error has been corrected in the most recent release of THETA (versions 1.7 and above).

6 Future Tasks

THETA has been a successful experiment in trusted, distributed operating system development. Below we provide a quick summary of the possible future tasks.

- Modify THETA to conform to emerging standards, namely the CORBA standard version 1.2. As part of this task, THETA must be improved in several cosmetic areas, which are described in Section 6.1.
- Address issues raised in the wide-area network study.
- Rework TNET
- Integrate a commercial, secure network facility into the THETA kernel.
- Fix replication.
- Replace dBase IV with COTS Trusted DBMS

6.1 Compliance with Standards

A major goal of THETA is to become compliant with emerging standards in the object-oriented world, in particular CORBA. Achieving this goal will make THETA far more usable, especially with the commercial world. As a part of this goal, we are participating in meetings to design the security framework for ORBs (Object Request Brokers) which will be compatible to the security policies and mechanisms in THETA.

As THETA becomes compatible and interoperable with commercial software, it must become more polished. General improvements must be made in several areas. For example:

- More administrator utilities need to be developed, and they should be more intuitive. Also, error reporting in the current set of tools needs to be improved to return more clear and concise diagnostic information.
- The functionality of THETA managers should be enhanced to make THETA services more responsive to the needs of users.
- The THETA manager skeleton should be reworked to better satisfy the TCSEC TCB minimization criteria.

- The THETA software libraries should be trimmed to remove obsolete functionality and redesigned to separate code needed for trusted operation from that needed for untrusted purposes.
- The user interfaces needs improvements. For example, most client applications are currently line-based where a graphical interface would be much nicer.
- Better configuration management tools and policies need to be put in place.
- Documentation, although it has been greatly improved in the last phase of this task, could still be more complete.

6.2 Secure Networks

THETA would benefit from using a commercial secure networking product. Removing network-dependent functionality from THETA will remove a few layers of complexity from the THETA kernel and would make the TCB smaller. We should survey the secure network products that are currently in evaluation and determine which would be most suited for THETA.

6.3 Replication

We must fix the database replication protocol that has been disrupted.

6.4 Scalable System

The wide area network (WAN) study looked into issues involved in running THETA over a wide area network versus its standard local area network environment. Some results of the WAN experiments have been addressed in this last phase of the project (e.g., improved network communication schemes). Several areas identified in the WAN study still need to be addressed. The list includes

- Develop security database analysis tools. Maintaining consistent security databases is a major concern for THETA WAN configurations. We wish to develop powerful tools to give administrators a clear view of THETA's static configuration and dynamic operation. Future THETA security analysis tools should be based on a DBMS and should support full consistency analysis of THETA and COS security databases.
- Revamp THETA auditing. To support a WAN environment, THETA should allow both centralized and distributed auditing. Also, the administrator would benefit greatly from an audit analysis application that would possibly incorporate some intrusion detection technology.
- Invocation tracking should be an added THETA feature. At present, THETA developers cannot determine the fate of timed-out invocations. Without this ability, developers will have great difficulty developing and debugging reliable THETA WAN applica-

tions. A Request Manager could be created to maintain status information about invocations and make it available to appropriate clients and managers.

- THETA should be adapted to permit separate administrative domains.

References

[1] Cheryl A. Barbasch. *AdaWise User's Manual*, July 1993. Prepared for the STARS contract.

[2] T. Bolognesi and H. Brinksma. Introduction to the ISO specification language LOTOS . *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[3] H. Brinksma. *The specification language LOTOS*. NGI, Amsterdam, 1985.

[4] Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DoD-5200.28-STD.

[5] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16:1058–1075, September 1990.

[6] David Guaspari, Mike Seager, and Matt Stillerman. Specifying the kernel of a secure distributed operating system. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, pages 285–306, Hemel Hempstead, 1995. Prentice Hall International Series in Computer Science.

[7] J. V. Guttag, J. J. Horning, and Andres Modet. Report on the larch shared language. Technical report, DEC/SRC, April 1990.

[8] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report TR 5, DEC/SRC, July 1985.

[9] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[10] D. F. Knuth. *Literate Programming*. Standford University, 1992.

[11] Daryl McCullough. Foundations of Ulysses: The theory of security. Technical Report RADC-TR-87-222, Rome Air Development Center, May 1988.

[12] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the Symposium on Security and Privacy*, pages 177–186, Oakland, CA, April 1988. IEEE.

[13] Daryl McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, June 1990.

[14] Object Management Group. Object Management Architecture Guide. OMG Doc. No. 92.11.1, OMG, 1992.

[15] Object Management Group. Object Services RFP3. OMG Doc. No. 94.7.1, OMG, 1994.

[16] Object Management Group. Universal Networked Objects. OMG Doc. No. 94.9.32, OMG, 1994.

[17] Object Management Group. Common Object Request Broker: Architecture and Specification. OMG Doc. No. 93.12.43, Revision 2.0, OMG, July 1995.

[18] ORA. Larch/Ada rationale. Technical report, ORA, September 1989.

[19] ORA. *Larch/Ada Reference Manual.* ORA, September 1989.

[20] ORA. Romulus Overview, March 1994.

[21] OSF. *OSF DCE User Guide and Refererence.* Prentice-Hall, 1993.

[22] D.S. Parker et al. Detection of mutual inconsistency on distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[23] Rita Pascale and Joseph McEnerney. Using THETA to implement access controls for separation of duties. In *Proceedings of the 17th National Computer Security Conference*, October 1994.

[24] Norman Proctor and Raymond M. Wong. The security policy of the secure distributed operating system prototype. In *Proceedings of the 5th Annual Aerospace Computer Security Applications Conference*, December 1989.

[25] Norman Ramsey. Literate programming simplified. *IEEE Software*, pages 97–105, September 1994.

[26] Norman Ramsey and Carla Marceau. Using literate programming in production. Technical report, ORA, July 1989.

[27] Norman Ramsey and Carla Marceau. Literate programming on a team project. Technical report, ORA, 1990, Software Practice and Experience.

[28] Deborah Russell and G. T. Gangemi. *Computer Security Basics.* O'Reilly and Associates, 1991.

[29] THETA staff. Computer System Operator's Manual for the THETA System. THETA CDRL No. A025, Odyssey Research Associates, Inc., April 1995.

[30] THETA staff. Descriptive Top Level Specification for the THETA System. THETA CDRL No. A029, Odyssey Research Associates, Inc., April 1995.

[31] THETA staff. Formal Security Policy Model for the THETA System. THETA CDRL No. A027, Odyssey Research Associates, Inc., April 1995.

[32] THETA staff. Manager Developer's Tutorial for the THETA System. THETA CDRL No. A034, Odyssey Research Associates, Inc., February 1995.

[33] THETA staff. Philosophy of Protection Report for the THETA System. THETA CDRL No. A028, Odyssey Research Associates, Inc., April 1995.

[34] THETA staff. Software Design Document for the THETA System. THETA CDRL No. A023, Odyssey Research Associates, Inc., April 1995.

[35] THETA staff. Software Requirements Specification for the THETA System. THETA CDRL No. A022, Odyssey Research Associates, Inc., April 1995.

[36] THETA staff. Trusted Computing Base Configuration Management Plan for the THETA System. THETA CDRL No. A031, Odyssey Research Associates, Inc., April 1995.

[37] THETA staff. Trusted Computing Base Verification Report for the THETA System. THETA CDRL No. A032, Odyssey Research Associates, Inc., April 1995.

[38] Trusted Information Systems, Inc. Demonstration Application User Orientation and Maintenance Guide for THETA Evaluation/Enhancements. Technical report, December 1994. This report was prepared by TIS under subcontract for the THETA project.

[39] Trusted Information Systems, Inc. THETA Trusted NETwork Communications Service Interface Design Document. THETA CDRL No. A004, April 1995. This report was prepared by TIS under subcontract for the THETA project.

[40] Trusted Information Systems, Inc. THETA Trusted NETwork Communications Service Interface Requirements Specification. THETA CDRL No. A003, April 1995. This report was prepared by TIS under subcontract for the THETA project.

[41] Trusted Information Systems, Inc. THETA Trusted NETwork Communications Service Operator Guide for THETA Evaluation/Enhancements. Technical report, April 1995. This report was prepared by TIS under subcontract for the THETA project.

[42] D. G. Weber and Robert S. Lubarsky. The SDOS project – Verifying hook-up security. In *Proceedings of Third Aerospace Computer Security Conference*, pages 7–15, Orlando, FL, December 1987. AIAA/ASIS/IEEE.

[43] John P. L. Woodward. Security requirements for system high and compartmented mode workstations. DDS-2600-5502-87, November 1987.

A DCE RPC Scenario

This appendix provides a step-by-step summary of the interactions between a client and a server of a DCE application, and the various DCE components that support secure client/server communication. The summary is in the form of a scenario in which there is a payroll application divided into client and server parts, distributed on different hosts so that payroll functions can be performed by individuals on various different workstations. The scenario consists of several hosts:

- Host *C* on which a user is running a payroll client program.
- Host *P* on which the payroll server is running.
- Host *D* on which a Cell Directory Server (CDS) is running.
- Host *S* on which a Security Server is running.

In addition, each host has an RPC server running on it, to maintain the endpoint map for that host.

The scenario begins when the client makes a procedure call to a client RPC stub. From then until the end of the scenario, all the events described herein take place "under the covers" of the RPC interface. These events are interactions between the DCE RPC runtime library of the client and the various servers.

After the client calls the RPC stub, the DCE runtime library (DRL) checks to see if it has a binding to a server for the UUID associated with the called RPC stub. This UUID is part of the state built into the client, to enable the client to use the CDS to locate a server that implements the RPC. In this case, the DRL finds no binding. This could be because this is the first RPC, or because the binding lapsed, perhaps due to network failure, host failure, or server failure. In this case, the reason is that this is the client's first RPC. Therefore, the steps below will also include the steps necessary to obtain credentials for the payroll server.

The first step in obtaining a binding to the payroll server is to find a host that is running a CDS. One method involves checking part of the configuration data of the client's host. In this case, part of Host *C*'s configuration is the information that Host *D* is running a CDS. In addition to the service UUID and the CDS host ID, there is one other piece of initial information that the client has: the fact that port 135 is the standard communication endpoint for any RPC

server on any host. This standard allows the RPC server to be bound to by any client without any further information, so that the client can then obtain RPC data about dynamic endpoint bindings for other servers.

Armed with this initial information, the DRL performs several bindings and RPCs.

1. Binding to port 135 on Host *D* to contact the RPC server on that host.
2. An RPC to the RPC server on Host *D* to obtain an endpoint for the CDS running on that host; the return value is in this case port 987.
3. Binding to port 987 on Host *D* to contact the CDS.
4. An RPC to the CDS to obtain the host ID of a host running the security server; the return value is *S*.
5. Binding to port 135 on Host *S* to contact the RPC server on that host.
6. An RPC to the RPC server on Host *S* to obtain an endpoint for the security server running on that host; the return value is in this case port 876.
7. Binding to port 876 on Host *S* to contact the security server.
8. A secure RPC to the security server to obtain credentials for access to the payroll server. This secure RPC uses a special previously obtained credential, in order obtain the client credential for the payroll server. (See below for more details).
9. An RPC to the CDS to obtain the host ID of a host running the payroll server; the return value is *P*.
10. Binding to port 135 on Host *P* to contact the RPC server on that host.
11. An RPC to the RPC server on Host *P* to obtain an endpoint for the payroll server running on that host; the return value is in this case port 234.
12. Binding to port 234 on Host *P* to contact the payroll server.

At this point, the role of the DRL is finished. The RPC stub has a binding and the appropriate credentials for the payroll server. The RPC stub then sends the RPC parameters, and waits for the RPC return data from the payroll server. When this arrives, the data is copied into the output parameters of the RPC stub procedure, which returns to the payroll client software. The payroll client software takes the output data and displays them on the screen for the user to see. Subsequent RPCs between the client and the server re-use the payroll credentials, rather than acquiring them again for every RPC.

With regard to security, there are several points to note. Firstly, credentials are on a per-server basis, and are sometimes called tickets, to indicate that each is for a specific server. A credential for one service does not grant access to another service.

Secondly, credentials are obtained from the security server, which itself requires a credential. This initial credential is used in RPCs to obtain other credentials, and is called a ticket-grant-

ing ticket (TGT). This is the special credential referred to in step 8 above. A TGT is most often obtained at the beginning of a user login session, and used throughout the session for any clients the user runs. This is why the TGT in step 8 was previously obtained, and can be used to obtain a ticket for payroll service in step 8.

Thirdly, none of the above-described location and endpoint RPCs are secure RPCs, i.e. contain no credentials. The host location and endpoint data are freely available with no restrictions on the acquisition of these data; no authentication or other message security is required. In this case, it is very important that neither the RPC server nor the CDS require credentials, because these servers are needed to obtain the location of the security server, in order to acquire credentials in the first place.

Finally, there are the details of the TGT acquisition, described in the next appendix.

B Summary of Ticket Acquisition and Usage

TGT acquisition is an RPC to the security server, which is done at the beginning of a user login session. As a result, this RPC is often called the DCE login. Another reason for this term is that the user must supply a password. When the DCE login is performed at the same time as the user's operating system login, then the DCE password is the same as the operating system login password. Although in this sense it is a password, the term is misleading because it is not used as a password by the security server.

The TGT-acquisition RPC is a non-authenticated RPC which requests a credential for the user on whose behalf the client is running, for example the user Mary. The security server returns Mary's TGT to the client. Note that any client can request Mary's TGT. However, only Mary's clients can use the credential, because it is encrypted with a secret key known only to Mary and to the security server. This key is specific to Mary, and was set up when Mary's account was created, or the last time Mary changed it by interacting with the security server.

Because Mary must remember her key and keep it secret to ensure that only she can successfully authenticate as Mary to the security server, the key must be easy to remember. As a result, the user actually remembers not a key but a word (often the same as her login password). This word is used as the basis for a computation to produce a DES key. Note that the keyword/password is used solely for local computation, and is never sent of over the network. It is not sent to the security server for a password comparison. Rather, it used to construct a secret shared with the security server, without sending the secret over the network. The basis of the authentication is *not* password comparison, but the possession of this shared secret, the key.

After performing this first RPC, the DRL performs cryptographic computation on the RPC return data sent from the security server. As mentioned above, the return data is encrypted with Mary's key. Therefore, the DRL uses Mary's password to construct the key, and then uses the key to decrypt the RPC return data. This decrypted RPC return data contains two items: the TGT, and a session key $K1$ to be used for future interaction between a Mary client and the security server.

The TGT itself is nothing more than the pair of $K1$ and Mary's ID, both encrypted by the security server's key. Therefore, future RPCs between a Mary client and the security server will include two things: the TGT, and the RPC parameter data encrypted with $K1$. When receiving such an RPC, the security server can decrypt Mary's TGT to obtain the session key $K1$, and use $K1$ to decrypt the RPC parameter data. The security server knows that message came from Mary, because only a Mary client could have obtained and used $K1$, which was sent as part of the return data of the TGT-acquisition RPC.

(Note: this account is actually a simplification, in that there is a second step involving another ticket and another session key, used to get a more complex kind of TGT, a PTGT. This PTGT and yet another session key are used for requests like that in step 8 above. However, in the interests of simplicity, we pass over these details. Also, the payroll server's authentication of Mary in involves more details than described below.)

Now we can better understand the payroll-ticket-acquiring RPC in step 8 above, and the subsequent use of the ticket. The client encrypts the payroll-ticket request with $K1$ and adds the TGT, to authenticate as Mary to the security server. The security server then sends back both a ticket for the payroll service, and also a session key $K2$ for Mary's client to use with the payroll service. The payroll service ticket is encrypted with the payroll server's key, and contains the following: Mary's ID, other authorization data such as Mary's group memberships, and a copy of $K2$.

Therefore, Mary can send authenticated RPCs to the payroll server by encrypting the RPC parameter data with $K2$, and adding the payroll ticket. The payroll server can decrypt the ticket using its own key, and thereby gain assurance that the ticket really came from the security server, which is the only other party that knows the payroll server's key. After decrypting the ticket, the payroll server has $K2$, and uses it to decrypt the RPC data. The ticket data identifies Mary as the caller. The payroll server has assurance that the RPC really came from Mary because only a Mary client could have received $K2$ from the security server, which embedded $K2$ in the payroll service ticket along with Mary's ID. Finally, the payroll server uses Mary's ID and groups to make an authorization decision about Mary's request.

C Acronyms

This section contains a list of acronyms and abbreviations used in the THETA documentation.

ACG	Automatic Code Generator
ACK	Acknowledgment
AFS	Andrew File System
AGS	Access Group Set
API	Application Programming Interface
BBN	Bolt, Beranek, and Newman, Incorporated
BLS	B Level Secure operating system
BOA	Basic Object Adapter
CDRL	Contract Deliverables Requirements List
CDS	Cell Directory Server
CMW	Compartmented Mode Workstation
CORBA	Common Object Request Broker Architecture
COS	Constituent Operating System
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSOM	Computer System Operator's Manual
CSU	Computer Software Unit
DAC	Discretionary Access Control
DCE	Distributed Computing Environment
DFS	Distributed File System
DRL	DCE Runtime Library
DID	Data Item Description
DoD	Department of Defense

DTLS	Descriptive Top Level Specification
HP-UX	Hewlett Packard UNIX operating system
IHP	Inter-host protocol
ID	Identifier
IDL	Interface Definition Language
IP	Inter-net Protocol
IPC	Inter-process Communications
ITC	Inter Task Communication
JDL	Joint Directors of Laboratories
KSP	Kernel Service Protocol
LAN	Local Area Network
MAC	Mandatory Access Control
MIP	Message in Progress
MLS	MultiLevel Secure
MSL	Multiple Single Level
NACK	Non-acknowledgment
NCSC	National Computer Security Center
NSA	National Security Agency
OMA	Object Management Architecture
OMG	Object Management Group
OP	Operation Protocol
ORA	Odyssey Research Associates, Inc.
ORB	Object Request Broker
OSF	Open Software Foundation
OSTF RFP3	Object Services Task Force Request For Proposal 3
PSL	Program Support Library (per manager)
RPC	Remote Procedure Call
SDD	Software Design Document
SEP	Security Evaluation Program
SLM	Single Level Manager
SPM	Software Programmer's Manual
SRS	Software Requirements Specification

SSDD	System Segment Design Document
SSS	System Segment Specification
STP	Security Test Plan
SUM	Software User's Manual
TCB	Trusted Computing Base
TCP/IP	Transmission Control Protocol / Internet Protocol
TCSEC	Trusted Computer System Evaluation Criteria
TGT	Ticket-Granting Ticket
THETA	Trusted Heterogeneous Architecture
TNI	Trusted Network Interpretation
UID	Unique Identifier
UNO	Unique Number
UUID	Universal Unique Identifier
WAN	Wide Area Network

D Glossary

This glossary is meant to offer the reader a quick definition to words that appear in THETA documentation. It is not intended to be exhaustive and can be supplemented by the glossary found in the TCSEC.

abstraction: A description of the external appearance of an entity that avoids defining and describing its internal details.

access control: The mechanisms used to protect objects from unauthorized access. Access control determines *who* may access an object and *how.*

access control list: An access control list (ACL) is a part of each THETA object's internal structure. An ACL is a set that specifies the rights that principals and groups have for accessing the object.

access group set: A THETA access group set (AGS) consists of a principal unique identifier and a list of group unique identifiers. The value of an AGS determines a principal's access rights.

ACL: See *access control list.*

AGS: See *access group set.*

ancestor: When used in the context of THETA types, ancestor refers to a type in the hierarchy between a given type and the root type *Object.* When used in the context of THETA groups, groups that contain other groups are referred to as ancestor groups.

asynchronous: Events that occur in an indeterminate order. Asynchronous operation invocations execute concurrently and complete in an arbitrary order.

atomic transaction: An action or set of actions that occur without interruption.

audit: Audit is the nickname for the Audit Manager, which is responsible for collecting information about security related events in THETA.

audit event: An audit event is a security related action that must be recorded by the Audit Manager.

audit log: This name is given to the Audit Manager's database, which is a collection of audit events.

authen: Authen is the name of the THETA manager that governs the objects used to grant discretionary access to THETA objects.

bindings: Each THETA process has several pieces of information *bound* to it as part of registration with the THETA kernel; these process bindings are the principal unique identifier associated with the COS user, the access group set unique identifier to be used for discretionary access control checks and the unique identifier of the process itself (which is assigned by the THETA kernel).

broadcast: An addressing mechanism that causes a message to be sent to all hosts on a network simultaneously.

canonical type: Canonical types provide a common format for data representation across all THETA hosts despite diverse internal representations of different machines. Canonical types are used in message construction, as operation argument and result types, and to define the characteristic form of data stored by objects of various THETA types. Note THETA types are not the same as canonical types. THETA types contain declarations of canonical types.

cantype: See *canonical type*.

client: A program or process that runs on behalf of a user and makes requests of other programs or processes, called servers. These *client* requests cause the *server's* process to perform some well-defined action called a service.

concurrent: Events that happen at the same time are said to be concurrent.

constituent operating system: This general term applies to any underlying operating system that supports THETA. Currently supported operating systems include Sun OS 4.1.X, Sun CMW 1.0, HP-UX 8.09, HP-UX BLS 8.09+, and AT&T System V/MLS.

core TCB: The THETA system is very flexible and can be configured to support multilevel or multiple single level processing. As a result, the trusted components vary depending on the configuration chosen. However, there is a core portion of these components that is always trusted, and this portion is called the *core TCB*. The core TCB is a subset of the THETA kernel and is identified specifically in the *Computer System Operator's Manual for THETA*.

COS: See *constituent operating system*.

Cronus: Cronus is the untrusted predecessor of THETA.

crosspoint: A Crosspoint is the name given to the shared memory-based communications implemented by UNIX-based THETA kernels.

DAC: See *discretionary access control.*

dir: Dir is the nickname of the Directory Manager that controls objects used to reference THETA objects by a symbolic means rather than the external representation of unique identifiers.

direct operation: A direct operation is THETA's implementation of distributed trusted path; this is detailed in the *Software Programmer's Manual for THETA*.

discretionary access control: This is a means of restricting access to objects based on lists of user identities and their access rights. The access control is considered discretionary since access rights are under the control of users rather than under the non-negotiable control of system labels (i.e., mandatory access control checks).

dominate: THETA level L1 is said to be dominated by THETA level L2 if and only if all categories of L1 are included in the set of categories of L2 and the hierarchical classification of L1 is less than or equal to that of L2.

downgrade: See *write down.*

edit: Data accesses that may require read and write access are known as "edits". If one opens a file for *edit,* one may *read* it, *write* it, or *do both. Edit* captures the combined idea expressed here better than "modify", which has the connotation of a required write or change.

generic object: An object that represents a type or the collection of objects of that type.

group: In THETA, a group is a list of principal unique identifiers and other group unique identifiers. Intuitively, a group is simply a list of THETA users.

host: A computer connected to the network.

host address: A 32-bit integer that uniquely identifies a host on the network.

incarnation number: A 24-bit integer that indicated the number of times that THETA has been brought up on a particular host. This number is one of the components of a unique number.

inheritance: New object types can adopt attributes from previously defined object types through *inheritance.*

instance: An instance refers to a specific, existing object.

internal representation: A data representation that is particular to a specific machine architecture.

invoke: To request the execution of an operation on an object.

kernel: The THETA kernel is a body of software that deals mainly with communications between THETA processes. It implements a trusted communications protocol (see kernel service protocol) that helps make the location of various THETA objects transparent to THETA applications.

kernel service protocol: The trusted communications protocol implemented by the THETA kernel. It is a session layer protocol that provides mandatory message security and supports THETA identification and authentication mechanisms.

key: THETA message construction and decomposition uses keys, which are numerical codes that THETA routines use as location markers in messages. The *Software Programmer's Manual for THETA* provides an extensive discussion of this concept.

KSP: See *kernel service protocol.*

levels: See *security levels.*

locate: An operation that searches the nodes of the network for the location of a particular object.

locator: A component of the THETA kernel that performs the *locate* request.

manager: A server process that handles operation invocations on objects that are in its (the manager's) domain.

MAC: See *mandatory access control.*

mandatory access control: This kind of access control uses security levels that are assigned to objects and users in order to mediate access.

messages: A THETA message is a sequence of key-cantype-value triples that provide both the structure and the information content of data transmitted between registered THETA processes.

migratory: A type is considered *migratory* if it is able to move among managers of the same object type on other hosts.

multicast: An addressing mechanism in which a given message can be sent to a group of logically related hosts.

node: Same as *host*. A computer connected to the network.

object: An abstract entity that can contain, transmit, or even process data. Normally, objects are thought of as structured repositories of data.

object cache: A list in the kernel that describes where objects with particular unique identifiers were found in the network most recently.

object database: An area on the local host's disk that stores the data values of objects of a particular type.

object manager: A process that regulates accesses to objects of one (or more) type(s) on a specific host.

object model: The conceptual framework in which all interactions are described by operations that are invoked on objects.

object type: The collection of all objects that have the same structure and have the same access mechanisms.

object-oriented system: A system that is built around an object model.

operation: An action involving objects performed by a server (or manager), on behalf of a client (application or manager).

persistent object: An object is considered *persistent* if the data is stored on some media (like in a file on a disk) and can be accessed again at some later point in time.

primal: Primal types are types whose objects have meaning only on the system where they were created. Primal objects must remain on the machine where they were created because that is the only machine where the data makes sense.

principal: A THETA principal is a COS user for whom a THETA authentication object is created and maintained. Data accesses are granted or denied depending on the principal and its access rights.

process: An executing program is referred to as a process.

program support library: This suite of routines is created by the autogeneration software to supply a client interface to manager operations. PSL calls insulate THETA programmers from the details of invocation message construction and reply message parsing. Each manager supplies new PSL routines.

PSL: See *program support library.*

read: Performing a read on an object is viewing the contents of a piece of data.

read down: In a secure system, a subject at a high security level can read data from a low level repository. This kind of read is permitted in the THETA security policy.

read up: In a secure system, a subject at a low level or incomparable level reads data from a high level repository. This kind of read is a forbidden in the THETA security policy.

readwrite: *Readwrite* refers to the combination of reading and writing actions performed on data, that is, editing. In a secure system, such editing demands that the subject and the object be at the same security level.

registration: This refers to the action taken by a COS process in order to be recognized by the THETA kernel.

registry: A registry in THETA is a special name space used in conjunction with authentication objects such as principals and groups. These name spaces are different from that supplied by the directory manager and associate symbolic names with unique identifiers.

replication: In distributed systems, this term refers to the act of copying or otherwise duplicating an object on a host other than the one on which it was created. Replication produces survivable application programs and facilitates recovery from data corruption.

restrictiveness policy: The restrictiveness policy is a formalization of the requirement that information in a multilevel secure system can flow only upward in security level.

rights: In THETA, this term refers to the discretionary access control privileges required to perform operations on objects. Rights are granted by a privileged user to other users or collections of users.

security labels: A sensitivity level assigned to a piece of information.

security levels: A combination of a hierarchical classification and a collection of categories that describe the sensitivity of information.

sequence number: A 24-bit unsigned integer that is increased by the kernel. This number is one of the components of a unique number.

server: A process that performs requests on behalf of other processes.

signature: A list of arguments and return values to a procedure, function, or operation.

synchronization: Ensuring that multiple copies of an object are kept consistent.

thfile: This is the nickname of the THETA File Manager.

transient object: An object is considered *transient* if it exists only for a brief period of time (for example, for the lifetime of the process that created it) and then disappears without a trace.

tropic: The **tropic** program is part of the THETA command set; **tropic** provides a universal client interface to all THETA managers. See the *Software User's Manual* for more details.

type definition: A type definition defines a class of objects in terms of data structures and operations that can be performed on those data structures.

type hierarchy: A tree-structured organization that shows the flow of type inheritance.

UID: See *unique identifier.*

unique identifier: In THETA, a unique identifier (UID) is a unique number (UNO) and a numerical type code that uniquely identifies an object in the THETA system. UIDs are used to identify THETA objects and in a very real sense should be considered to be an *address.*

unique number: In THETA, a UNO refers to a 14 byte long integer. A UNO can be decomposed into subfields that *may* provide information about THETA kernel incarnation and sequence numbers, a host address, and what security level is being referenced. The host address is 32 bits and can be used to encode the Internet address in future versions of the system, if need be. UNOs are the main constituent of unique identifiers.

UNO: See *unique number.*

upgrade: The THETA kernel raises the security level of some messages based on the level of the target object in the message and the level of the sender.

user interface: A collection of programs and subprograms organized to give a user a means to interact with THETA.

write: Any action by a subject that changes the contents of an object is considered to be a *write* to that object. In addition, any action that affects any *other* user's view of an object must also be considered a *write.*

write down: In secure systems, the act of *writing* data by a process at a high security level to an object at a lower or incomparable level is considered to be a *write down.* This downgrading of information is forbidden in THETA.

write up: In secure systems, the act of *writing* data by a process at a low security level to an object at a higher level is considered to be a *write up.* This upgrading is permitted by THETA security policy.

Index

*U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61110

MISSION OF ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.